RAND MODEL DESIGNER

Version 6.0

USER MANUAL

© 2010 Rand-Service Ltd. All rights reserved.

The information contained herein is proprietary to Rand-Service Ltd. and shall not be duplicated in whole or in part. The technical details contained in this manual are the best that are available at the date of issue but are subject to change without notice.

Rand-Service Ltd. pursues the policy of continuous development. This may lead to the product described in this manual being different from the product delivered after its publication.

Contents:

CHAPTER 1 Introduction 10
1.1 What Is Rand Model Designer? 11
1.2 How Rand Model Designer Works 11
1.3 Document Overview 13
CHAPTER 2 Modeling Language 14
2.1 Project 15
2.2 Classes 16
2.3 Stereotype 17
2.4 Behavior Chart 17
2.4.1 State 18
2.4.2 Transition 19
2.4.3 Internal Transition 20
2.4.4 Initial State 20
2.4.5 Final State 21
2.4.6 Switch Point 21
2.4.7 Probabilistic Switch Point 22
2.5 System of Equations 22
2.6 Aggregate System of Equations 26
2.7 Structure Diagram 27
2.7.1 Local Object 27
2.7.2 Class Boundary 28
2.7.3 Link 29
2.8 Parameters 33
2.9 Variables 33
2.10 Constants 35
2.11 Local Classes 35
2.12 Class Inheritance 36
2.13 Types 36
2.13.1 Scalar Types 37
2.13.1.1 double 37
2.13.1.2 byte, short, integer 38
2.13.1.3 boolean 39
2.13.1.4 char 39
2.13.1.5 string 39

2.13.1.6 Enumerations 39
2.13.2 Regular Types 40
2.13.2.1 Vectors 40
2.13.2.2 Matrices 40
2.13.3 Record 41
2.13.4 Connector 42
2.13.5 Signal 42
2.13.6 Declared Types 43
2.14 Functions and Procedures 43
2.14.1 Predefined Functions and Procedures 44
2.14.1.1 Standard Math Functions 44
2.14.1.2 Functions and Procedures for Vectors and Matrices 45
2.14.1.3 Random Number Generators 46
2.14.1.4 Special Functions and Procedures 46
2.14.1.5 Functors 48
2.14.1.5.1 integral 48
2.14.1.5.2 maxfun 48
2.14.1.5.3 minfun 48
2.14.1.5.4 root 49
2.14.2 User-Defined Functions and Procedures 49
2.14.2.1 Internal Functions and Procedures 49
2.14.2.2 External Functions and Procedures 51
2.15 Expressions 54
2.15.1 Operations 54
2.15.2 Evaluation Order 55
2.16 Operators 56
2.16.1 Assignment Operator 56
2.16.2 Procedure Call Operator 57
2.16.3 The if Operator 57
2.16.4 The case Operator 58
2.16.5 The loop Operator 58
2.16.6 The exit Operator 59
2.16.7 The return Operator 60
2.16.8 The send Operator 60
2.16.9 Optimization Operator 61

2.17 Visibility Rules 61
CHAPTER 3 Integrated Modeling Environment 63
3.1 Main Window 64
3.2 Working with Projects 64
3.3 Project Manager 67
3.3.1 Classes 68
3.3.2 Declared Types 69
3.3.3 Constants 71
3.3.4 User-Defined Functions and Procedures 73
3.3.5 Deleting Project Components 73
3.3.6 Importing and Exporting Project Components 74
3.4 Class Editor 75
3.4.1 Class Editor Tree 75
3.4.1.1 Class Name 76
3.4.1.2 Class Icon 76
3.4.1.3 Superclass 77
3.4.1.4 Parameters 79
3.4.1.5 Variables 82
3.4.1.6 Constants 86
3.4.1.7 Functions and Procedures 88
3.4.1.8 Local Classes 89
3.4.1.9 Local Objects 89
3.4.1.10 Comments 90
3.4.1.11 Deleting Class Elements 91
3.4.1.12 Modifying Class Stereotype 91
3.4.2 Equation System Editor 91
3.4.3 Behavior Chart Editor 98
3.4.3.1 Toolbar Buttons 98
3.4.3.2 Editing Area 99
3.4.3.3 States 101
3.4.3.4 Transitions 105
3.4.3.5 Special States 109
3.4.4 Structure Diagram Editor 109
3.4.4.1 Toolbar Buttons 110
3.4.4.2 Editing Area 111

3.4.4.3 Local Objects 112
3.4.4.4 Class Boundary 115
3.4.4.5 External Variables 115
3.4.4.6 Links 117
3.4.5 The Model Class Features 118
3.5 The “Choose type” Dialog Box 120
3.6 The “Groups” Dialog Box 126
3.7 Text Editor 127
3.7.1 Toolbar Buttons 128
3.7.2 Editing Area 129
3.8 Component Editor 129
3.9 Calculator 130
3.10 Textual Project Representation 132
3.11 Working with Executable Models 135
3.12 The “Project options” Dialog Box 136
3.12.1 The “General” Tab 137
3.12.2 The “Folders” Tab 138
3.12.3 The “Model” Tab 139
3.12.4 The “Version” Tab 140
3.13 The “Environment options” Dialog Box 140
3.13.1 The “General” Tab 141
3.13.2 The “Colors” Tab 141
3.13.3 The “Display” Tab 142
3.13.4 The “Fonts” Tab 143
3.14 Miscellaneous Functions 143
CHAPTER 4 Executable Model 145
4.1 Executable Model Categories 146
4.2 Visual Executable Model 146
4.2.1 The “Variables” Window 147
4.2.1.1 Virtual Variables 149
4.2.2 The “Behavior Chart” Window 150
4.2.3 The “Structure” Window 153
4.2.4 The “Diagram” Window 157
4.2.4.1 The “Diagram parameters” Dialog Box 163
4.2.5 The “Histogram” Window 165

4.2.6 The “2D Animation” Window 172
4.2.6.1 The “Standard 2D components” Panel 173
4.2.6.2 Using the “2D Animation” Window 174
4.2.6.3 Dancer 175
4.2.6.4 Linear Indicator (Continuous or Discrete) 176
4.2.6.5 Slider 176
4.2.6.6 Color Indicator 176
4.2.6.7 Sprite 177
4.2.6.7.1 The “Animation parameters” Dialog Box 178
4.2.6.8 Diagram 179
4.2.6.9 Digital Indicator 179
4.2.6.10 Knob 179
4.2.6.11 Button 179
4.2.7 The “3D Animation” Window 180
4.2.7.1 The “3D animation parameters” Dialog Box 181
4.2.7.1.1 The “General” Tab 182
4.2.7.1.2 The “3D objects” Tab 182
4.2.7.2 Using the “3D Animation” Window 183
4.2.7.3 Line Segment 185
4.2.7.4 Sphere 185
4.2.7.5 Cylinder 186
4.2.7.6 Cone 186
4.2.7.7 Torus 187
4.2.7.8 Spring 187
4.2.7.9 Beam 187
4.2.7.10 Quadrangle 188
4.2.7.11 Arrow 188
4.2.7.12 Surface 188
4.2.7.13 Text Label 189
4.2.8 The “Aggregate System of Equations” Window 189
4.2.9 The “Equation Blocks” Window 191
4.2.10 The “Calculator” Window 191
4.2.11 Model Time 193
4.2.12 The “Breakpoint conditions” Dialog Box 193
4.2.12.1 The “Time” Tab 194

4.2.12.2 The “States” Tab 194
4.2.12.3 The “Transitions” Tab 195
4.2.12.4 The “Predicates” Tab 196
4.2.13 Model Settings 197
4.2.13.1 The “Model settings” dialog box 198
4.2.13.1.1 The “Run” Tab 199
4.2.13.1.2 The “View” Tab 200
4.2.13.1.3 The “Files” Tab 200
4.2.13.2 The “Numerical methods” Dialog Box 201
4.2.13.2.1 The “Tolerance” Tab 202
4.2.13.2.2 The “Differential equations” Tab 203
4.2.13.2.3 The “Algebraic-differential equations” Tab 204
4.2.13.2.4 The “Algebraic equations” Tab 205
4.2.13.2.5 The “Optimization” Tab 206
4.2.13.2.6 The “Options” Tab 207
4.2.14 Executing the Model 207
4.2.15 The “Simulation Plan” Dialog Box 208
4.2.16 Saving and Restoring the Current Model State 211
4.2.17 Miscellaneous Functions 212
4.2.18 Using the Standalone Visual Executable Model 213
4.3 Embeddable Executable Model 213
4.3.1 Data Types and Constants 214
4.3.2 Model Creation and Destruction Functions 215
4.3.3 Functions to Save and Restore Model State 215
4.3.4 Model Execution Functions 216
4.3.5 Variable Management Functions 217
4.3.6 Numerical Methods Configuration Functions 219
4.3.7 Miscellaneous Functions 220
CHAPTER 5 SysLib, The Standard Class Library 222
5.1 Linear Blocks 223
5.1.1 CGain 223
5.1.2 CSummator 223
5.1.3 CSubtractor 224
5.1.4 CIntegrator 224
5.1.5 CDifferentiator 225

5.1.6 CAperiodicUnit 225
5.1.7 COscillatingUnit 226
5.2 Nonlinear Blocks 227
5.2.1 CDeadZone 227
5.2.2 CSaturation 228
5.2.3 CRelay 229
5.2.4 CSwitch 230
5.2.5 CInterpolator 231
5.3 Signal Sources (Generators) 231
5.3.1 CRampGenerator 231
5.3.2 CSineGenerator 232
5.3.3 CSawtoothGenerator 233
5.3.4 CPulseGenerator 234
5.3.5 CStepsGenerator 235
5.3.6 CClock 235
CHAPTER 6 Appendices 236
6.1 Extended Backus-Naur Form 237
6.2 Identifier 237
6.3 The MVL Language 238

CHAPTER 1
Introduction

Introduction to Rand Model Designer
and this document

1.1 WHAT IS RAND MODEL DESIGNER?

Rand Model Designer is a simulation modeling tool that allows the user to create and experiment with models of complex dynamic systems.

The Rand Model Designer's key features are:

Object-oriented modeling language

Intuitive and easy-to-use integrated modeling environment

Ability to create continuous, discrete, and hybrid (continuous-discrete) simulation models

Use of differential and algebraic equations to model continuous behavior

Use of behavior charts (simplified statecharts) to model discrete behavior

Model correctness verification throughout the model creation process

Automatic generation of the executable model from the model description created by the user

Interactive and automated experiments with the executable model

2D and 3D animation

Support for creation and re-use of libraries of model components

Generation of the executable model as either a standalone Windows application or a dynamic link library

1.2 HOW RAND MODEL DESIGNER WORKS

The user starts using Rand Model Designer by opening the *integrated modeling environment*. The integrated modeling environment is a Windows application that looks as shown in **Figure 1**.

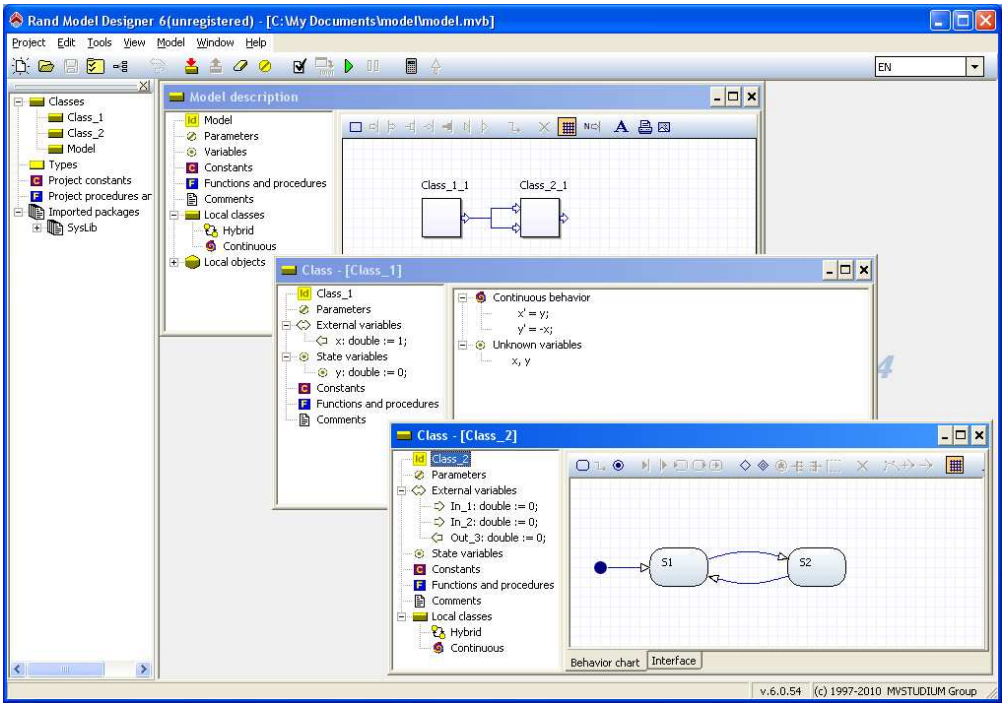


Figure 1 Integrated modeling environment

To begin authoring a model, the user creates a *project* in the integrated modeling environment. A project is a model description consisting of model components. The user populates the project with components as necessary, with the goal of creating an adequate model of the system being modeled.

When the model description is ready, the user *runs* the model by clicking the **Run model** toolbar button or choosing the respective item from the main menu. This automatically generates and starts the *visual executable model*. The visual executable model is a Windows application that implements the model and allows the user to experiment with it. An example of the visual executable model is shown in **Figure 2**.

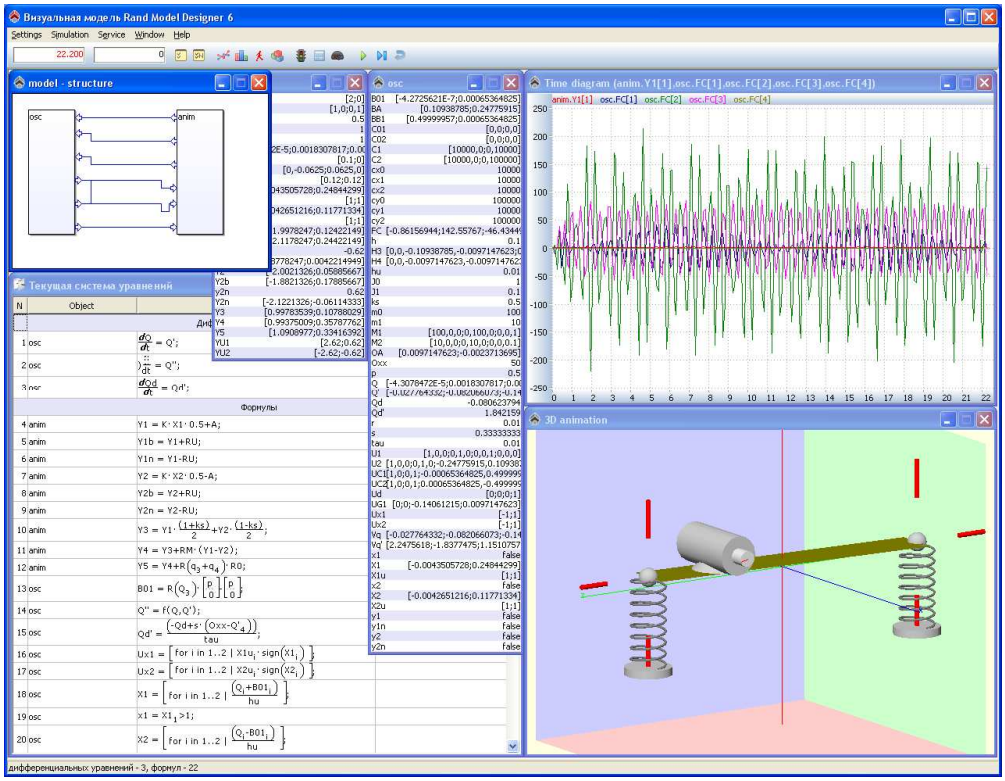


Figure 2 Visual executable model example

At any moment, the user can close the visual executable model, get back to the integrated modeling environment, make changes to the project, and run the model again.

Rand Model Designer models are built from components. Within a particular project, the user can utilize both the components defined in the project and the components found in available *libraries*. Libraries are collections of model components created by the same user or other users of Rand Model Designer. The Rand Model Designer distribution includes one readily available library: SysLib, the standard class library.

Should the user have a need to integrate the executable model with other software on the user side, the *embeddable executable model* can be created instead of the visual one. The embeddable executable model is a Windows DLL that any software can link to in order to get programmatic access to the model.

1.3 DOCUMENT OVERVIEW

This document is organized as follows:

Chapter 2, **Modeling Language**, describes the modeling language of Rand Model Designer.

Chapter 3, **Integrated Modeling Environment**, describes the integrated modeling environment.

Chapter 4, **Executable Model**, discusses the categories and capabilities of executable models.

Chapter 5, **SysLib, The Standard Class Library**, serves as a reference manual for the SysLib library.

Chapter 6, **Appendices**, contains appendices.

CHAPTER 2
Modeling Language

This chapter describes the modeling language
of Rand Model Designer

2.1 PROJECT

A *project*, on one hand, provides a complete definition of a model and, on the other hand, is a collection of components each of which may or may not be a part of the model definition. A project corresponds to a package in UML notation.

A project can contain the following components:

Classes

Declared types

Project constants

Project functions and procedures

Imported packages

Classes are the primary components of every project. Classes are discussed in **Section 2.2** of this document.

In addition to classes, a project can contain any number of *declared types*, *constants*, *functions*, and *procedures*. Declared types and constants are discussed in **Section 2.13.6** and **Section 2.10**, respectively. Functions and procedures are discussed in **Section 2.14.2**.

Every project contains a special class called `Model`. The `Model` class is special in the sense that it is interpreted as the complete definition of the model. Rand Model Designer creates an instance of the `Model` class automatically upon executable model startup and uses it to drive the execution of the model. The fact that every project contains the `Model` class means that there is a one-to-one relationship between projects and model definitions. Every project provides a definition of some model, while every model definition resides in some project.

Each component of a project has *visibility*, which can be either *private* or *public*. (Declared types always have public visibility.) A component with private visibility – a private component – can only be used in its containing project. A component with public visibility – a public component – can be used in both its containing project and other projects.

In order to be able to use a public component of another project, a project must *import* that other project. An imported project is called an *imported package*. Every project has an associated list of imported packages that identifies the packages imported into the project. All public components of an imported package are imported into the importing project, that is, become available for use in the importing project. The importing project can refer to imported components using compound component names that include the imported package name as the prefix, for example, `SysLib.CSineGenerator`.

A project whose public components are mainly intended for use in other projects is usually called a *library*. Even though the contents of a library is meant for use in other projects, every library has the `Model` class just like any other project. In libraries, the `Model` class is usually set to be private and have either some contents aimed to test the library or no important contents at all.

2.2 CLASSES

A *class* is a modeling language construct that defines the characteristics that the objects of the class all share. A class is fully specified by a class definition. In the most general case, a class definition consists of:

Name

Icon

Comments

Stereotype

Behavior chart (if the class is hybrid)

System of equations (if the class is continuous)

Structure diagram (if the class is compound and/or open)

Parameters

Variables

Class constants

Class functions and procedures

Local classes (if the class is hybrid)

Base class (if the class inherits from another class)

Indication whether the class is concrete or abstract

The *name* of a class is an identifier unique across all classes of the project. The syntax of identifiers is described in **Section 6.2**.

The *icon* of a class is an optional image shown on every instance of the class whenever instances of the class appear visually.

The *comments* are text associated with the class. The comments have no defined syntax or semantics.

The *stereotype* defines the behavior type, the structure type, the component type, and the visibility of the class. The stereotype is discussed in detail in **Section 2.3**.

Depending on the stereotype, the behavior of a class is specified by either the *behavior chart* or the *system of equations*. The behavior chart is described in **Section 2.4**. The system of equations is discussed in **Section 2.5**.

The *structure diagram* defines the internal structures and the interface of the class. The structure diagram is discussed in **Section 2.7**.

A class can have an arbitrary number *of parameters* and *variables*. Parameters are discussed in **Section 2.8**. Variables are discussed in **Section 2.9**.

Like a project, a class can contain an arbitrary number of *constants*, *functions*, and *procedures*. Constants are discussed in **Section 2.10**. Functions and procedures are discussed in **Section 2.14.2**. The constants, functions, and procedures of a class differ from the constants, functions, and procedures of a project in their *scope*, see **Section 2.17**. The constants, functions, and procedures of a project are visible in the entire project. Moreover, the public constants, functions, and procedures of a project are visible in all other projects that import their containing project. The constants, functions, and procedures of a class are only visible within their containing class.

A class can contain an arbitrary number of *local classes*. Local classes are used to associate behavior with states of the behavior chart. Local classes are discussed in **Section 2.11**.

If a class has a *base class*, it *inherits* all characteristics of the base class. Class inheritance is discussed in **Section 2.12**.

A class can be declared as either *concrete* (the default) or *abstract*. Abstract classes differ from concrete classes in that creation of instances of abstract classes is not allowed, whereas creation of instances of concrete classes is allowed. Abstract classes can only be used as base classes. A class that inherits from an abstract class can in its turn be either concrete or abstract.

2.3 STEREOTYPE

The *stereotype* of a class determines the behavior type, the structure type, the component type of the class, and the ability to use the class in other projects.

The stereotype is a combination of the following characteristics:

Behavior type

Structure type

Component type

Visibility

The *behavior type* indicates that the class has either *continuous behavior*, *hybrid behavior*, or no behavior. A class that has continuous behavior is called a *continuous class*. The behavior of a continuous class is specified by the *system of equations*, see **Section 2.5**. A class that has hybrid behavior is called a *hybrid class*. The behavior of a hybrid class is specified by the *behavior chart*, see **Section 2.4**.

The *structure type* indicates that the class is either *elementary* or *compound*. An elementary class does not have internal structure. A compound class does have the internal structure. The internal structure of a compound class is specified by the structure diagram, see **Section 2.7**.

The *component type* indicates that the class is either *isolated* or *open*. An isolated class does not have interface and thus cannot interact with the environment. An open class does have the interface and, therefore, can interact with its environment. The interface of an open class is specified by the structure diagram, see **Section 2.7** and **Section 2.7.2**.

The *visibility* indicates that the class is either *private* or *public*. A private class can only be used in its containing project. A public class can be used in any project. For the detailed discussion of the visibility of project components, see **Section 2.1**.

2.4 BEHAVIOR CHART

If a class is hybrid (see **Section 2.3**), it has the associated *behavior chart*. A behavior chart is a diagram that defines the behavior of a hybrid class. A behavior chart consists of *states* and *transitions* between states. At any time moment, exactly one state of a behavior chart is said to be *current*. A behavior chart corresponds to a UML statechart without composite and orthogonal states.

2.4.1 State

A *state* represents some lasting state of the modeled object. On a behavior chart, a state is shown as a rectangle with rounded corners, see **Figure 3**.

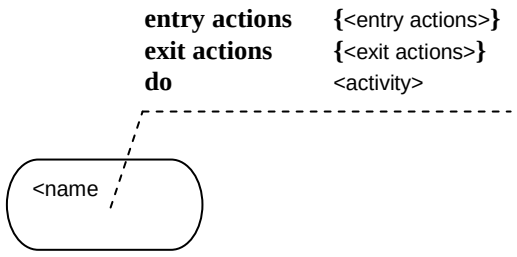


Figure 3 State

A state has the following properties:

Name

Entry actions

Exit actions

Activity

The *name* of a state is an identifier (see **Section 6.2**) unique across all states of the behavior chart.

The *entry actions* of a state are a sequence of operators (see **Section 2.16**) that is executed instantly every time the state is entered. A state may have no entry actions, in which case no actions are executed when the state is entered.

The *exit actions* of a state are a sequence of operators that is executed instantly every time the state is exited. A state may have no exit actions, in which case no actions are executed when the state is exited.

The *activity* of a state is specified by a class and a set of actual parameter values (see **Section 2.8**) and initial variable values (see **Section 2.9**) for an instance of that class. The activity is executed while the state remains current. The class used to specify the activity is called the activity class. An instance of the activity class is called an activity object. An activity object is created automatically when the state is entered, executes the state's activity while the state remains current, and is destroyed automatically when the state is exited. The variables of an activity object are accessible from the state's entry and exit actions and from the actions of transitions existing the state. Any ordinary class available in the project or local class defined in the containing class (see **Section 2.11**) can be used as the activity class. If the activity class is continuous, it is said to implement a continuous activity. If the activity class is hybrid, it is said to implement a hybrid activity. On a behavior chart, a state with a continuous activity is shown with a thick border as in **Figure 4**. A state with a hybrid activity is shown with a double border as in **Figure 5**.

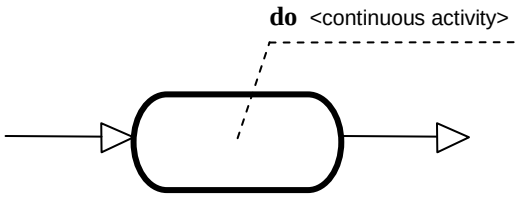


Figure 4 Continuous activity state

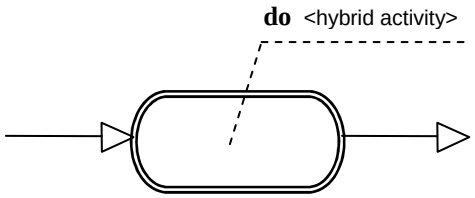


Figure 5 Hybrid activity state

A state may have no associated activity, in which case no activity is executed while the state remains current.

2.4.2 Transition

A *transition* is shown as an arrow starting at the border of one state, called the *source state*, and terminating at the border of another or possibly the same state, called the *target state*, see **Figure 6**.

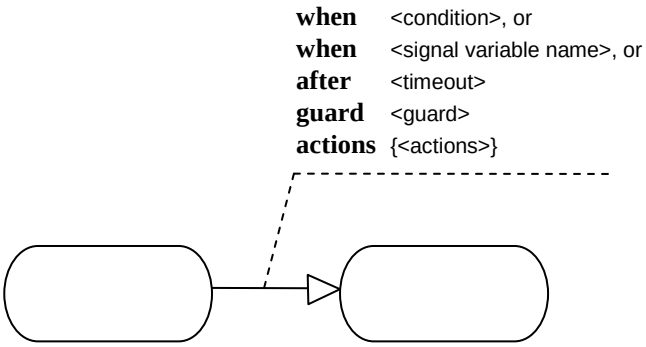


Figure 6 Transition

A transition has the following properties:

Trigger

Guard

Actions

The presence of a transition on a behavior chart indicates that if the specified trigger occurs while the object is in the transition's source state, the object will instantly transition to the target state and execute the specified actions, but only if the specified guard is satisfied.

The *trigger* of a transition can be specified in one of the following ways:

when <condition> – the trigger occurs if and when the specified condition becomes `true`. The condition is specified by an expression that evaluates to `boolean` (see **Section 2.15**).

when <signal variable name> – the trigger occurs if and when the specified variable receives a signal (see **Section 2.13.5** and **Section 2.16.8**).

after <timeout> – the trigger occurs if and when the time spent in the current state exceeds the specified timeout. The timeout is specified in model time units by an expression that evaluates to `double` (see **Section 2.15**).

A transition may have no trigger. If a transition has no trigger, the trigger occurs immediately after the object enters the transition's source state.

The *guard* of a transition is specified by an expression that evaluates to `boolean` (see **Section 2.15**). A transition is executed if its guard evaluates to `true` at the time the trigger occurs. If the guard does not evaluates to `true` at the time the trigger occurs, the transition is not executed and the trigger occurrence is lost. A transition may have no guard, in which case the guard is assumed to always be `true`.

A transition that has neither trigger no guard is called an *unconditional transition*. An unconditional transition is executed immediately after the object enters the transition's source state.

The *actions* of a transition are a sequence of operators (see **Section 2.16**) that is executed instantly every time the transition is executed. A transition may have no actions, in which case no actions are executed when the transition is executed.

2.4.3 Internal Transition

An *internal transition* is a transition of a special kind that takes place within a state without changing the current state. Unlike an ordinary transition, an internal transition is not shown as a separate arrow on a behavior chart. Instead, a state containing one or more internal transitions is shown as a rectangle with an arrow inside, as in **Figure 7**.

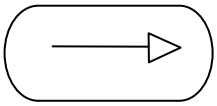


Figure 7 State with internal transition(s)

When an internal transition is executed, the object does not leave the current state. The execution of an internal transition, therefore, does not result in execution of the current state's exit and entry actions or any changes to the state's activity. This is what makes an internal transition different from an ordinary transition exiting and entering the same state. The execution of an ordinary transition exiting and entering the same state results in execution of the state exit actions, destruction of the current state activity object, creation of a new state activity object, and execution of the state entry actions.

In all other respects internal transitions are equivalent to ordinary transitions.

2.4.4 Initial State

An *initial state* is a special state from which the execution of a behavior chart begins. An initial state is shown as a solid filled circle, see **Figure 8**.

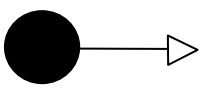


Figure 8 Initial state with mandatory outgoing transition

Every behavior chart must have exactly one initial state. Every initial state must have exactly one outgoing transition, which must be unconditional. An initial state cannot have incoming transitions.

An initial state cannot have entry actions, exit actions, or activity.

2.4.5 Final State

A *final state* is a special state that terminates the execution of a behavior chart. A final state is shown as a circle surrounding a smaller solid filled circle, see **Figure 9**.

Figure 9 Final state

A final state cannot have outgoing transitions.

A final state cannot have entry actions, exit actions, or activity.

2.4.6 Switch Point

A *switch point* is a special state that, when visited, results in immediate selection and execution of one of the outgoing transitions. On a behavior chart, a switch point is shown as a rhomb, see **Figure 10**.

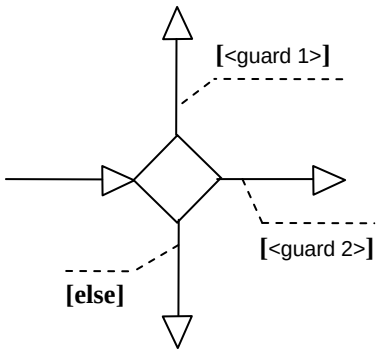


Figure 10 Switch Point

A transition exiting a switch point cannot have a trigger. A transition exiting a switch point can have either a guard or a special **else** label. A transition with an **else** label is called an *else-transition*. A switch point can have at most one outgoing else-transition.

When a switch point is visited, an outgoing transition whose guard evaluates to `true` is immediately selected and executed. If there is no such transition, the else-transition is executed.

If a switch point does not have an outgoing else-transition, upon visiting the switch point the guard of exactly one outgoing transition must evaluate to `true`. If a switch point does have an outgoing else-transition, upon visiting the switch point the guard of at most one outgoing transition may evaluate to `true`.

A switch point cannot have entry actions, exit actions, or activity.

2.4.7 Probabilistic Switch Point

A *probabilistic switch point* is a special state that, when visited, results in immediate selection and execution of one of the outgoing transitions. The transition to be executed is selected randomly among the outgoing transitions based on the probabilities associated with them. On a behavior chart, a probabilistic switch point is shown as a rhomb surrounding a smaller solid filled rhomb, as in **Figure 11**.

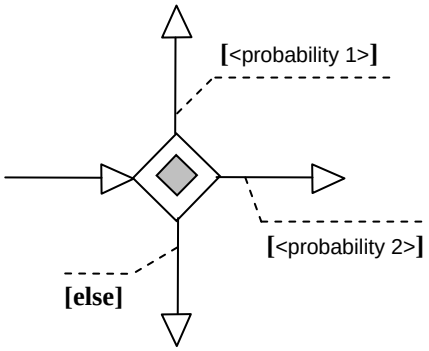


Figure 11 Probabilistic switch point

A transition exiting a probabilistic switch point can have neither trigger nor guard. Instead of a guard, a transition exiting a probabilistic switch point can have either a *probability* or a special **else** label. The probability is specified by an expression (see **Section 2.15**) that evaluates to `double`. A transition with an **else** label is called an *else-transition*. A probabilistic switch point can have at most one outgoing else-transition.

The probability of a transition indicates the probability of selection of that transition upon a visit to the probabilistic switch point. If a probabilistic switch point has an outgoing else-transition, its probability is assumed to be equal to $1 - S_r$, where S_r is the sum of the probabilities of other transitions exiting the probabilistic switch point.

If a probabilistic switch point does not have an outgoing else-transition, upon visiting the probabilistic switch point the sum of the probabilities of all outgoing transitions must be equal to `1`. If a probabilistic switch point does have an outgoing else-transition, upon visiting the probabilistic switch point the sum of the probabilities of other outgoing transitions must be less than or equal to `1`.

A probabilistic switch point cannot have entry actions, exit actions, or activity.

2.5 SYSTEM OF EQUATIONS

If a class is continuous (see **Section 2.3**), it has the associated *system of equations* that defines the behavior of the class. A system of equations consists of:

One or more equations

Zero or more declarations of known and unknown variables

A set of the initial values of the first derivatives of those variables whose second derivatives participate in the equations

Every equation has the following form:

```
<expression 1> = <expression 2>
```

where <expression 1> and <expression 2> are expressions (see **Section 2.15**) that can contain the standard operations (see **Section 2.15.1**), variables (see **Section 2.9**), parameters (see **Section 2.8**), constants (see **Section 2.10**) and functions (see **Section 2.14**) visible in the class, as well as the first and second derivatives of the variables with respect to time. In equations, the first derivative of x with respect to time is written as $\frac{dx}{dt}$, the second – as $\frac{d^2x}{dt^2}$ (see **Section 3.7**).

Examples of individual equations (x, y and z are variables, k and p are parameters):

$x = 0$

$x + y = 0$

$x = y$

$x = y + z$

$x^2 = y^2 + z^2$

$x + y = k \cdot e^z$

$x = \sin(Time)$

$x \cdot y = \sin(p \cdot z)$

$sqrt(x) - x = function_1(y)$

$\frac{dx}{dt} = -x$

$\frac{dx}{dt} = -x + y$

$\frac{dx}{dt} = z + \frac{dy}{dt}$

$\frac{dx}{dt} + \frac{dy}{dt} + \frac{dz}{dt} = 0$

$\left(\frac{dx}{dt}\right)^2 = \sin\left(\frac{dy}{dt}\right) + y$

$\frac{d^2x}{dt^2} = -x$

$\frac{d^2x}{dt^2} + \frac{dx}{dt} + x = 0$

$\frac{d^2x}{dt^2} + \frac{dx}{dt} + x = \left(\frac{d^2y}{dt^2} + \frac{dy}{dt} + y\right)^2$

The system of equations of a class can contain one or more equations, for example:

$\frac{d^2x}{dt^2} = -x$

or

$$\frac{dx}{dt} = y$$
$$\frac{dy}{dt} = -k \cdot x$$

or

$$x + y = 0$$
$$x - y = k$$

or

$$x^2 = y^2 + z^2$$
$$\frac{dx}{dt} = 3 - y$$
$$y - z = k$$

The set of variables visible in a compound class (see **Section 2.3**) includes external variables of the compound class' local objects (see **Section 2.7.1**). External variables of local objects can, therefore, participate in the system of equations of a compound class. In the system of equations of a compound class, an external variable of a local object can be referred to as

```
<local object name>.<variable name>
```

For example, if a class contains a local object `obj` with an external variable `var`, the system of equations of the class can refer to that variable as `obj.var`:

$$\frac{dobj.var}{dt} = x - obj.var$$

$$x + obj.var = 0$$

Besides equations, a system of equations can contain an arbitrary number of declarations indicating that some variables participating in the equations should be considered unknown and some should be considered known. *Unknown variables* represent unknown values that need to be found by solving the equations. *Known variables* are those variables that participate in the equations but are not unknown. Known variables are either fixed values or known functions of other known variables and/or time.

A declaration indicating that one or more variables are unknown has the following syntax:

```
unknown <variable name>, ...
```

For example:

```
unknown x
```

or:

```
unknown x, y, obj.var
```

A declaration indicating that one or more variables are known has the following syntax:

```
known <variable name>, ...
```

For example:

```
known x
```

or:

```
known x, y, obj.var
```

If the number of variables declared unknown is less than the number of equations in the system of equations (this includes the case when no variables are declared unknown), Rand Model Designer attempts to automatically select the required number of variables and declare them unknown to equalize the number of unknown variables with the number of equations. Rand Model Designer selects unknown variables from the set of *free variables*. Free variables are variables that participate in the system of equations and are not yet declared as either known or unknown.

If the number of free variables is equal to the number of unknown variables that Rand Model Designer needs to select, the selection succeeds and all free variables become unknown. The resulting system of equations is considered *formally resolvable*. For example, given the system of equations

$$x + y = 1$$
$$x - y = 0$$

(where x and y are variables) Rand Model Designer selects x and y as unknown variables and considers the resulting system formally resolvable. Likewise, given the system of equations

$$x + y = 0$$
$$y + x = 0$$

Rand Model Designer selects x and y as unknown variables and considers the resulting system formally resolvable too.

If the number of free variables is greater than the number of unknown variables that Rand Model Designer needs to select, the automatic selection fails, the system of equations is considered *underdetermined*, all free variables remain free, and the selection of unknown variables is deferred to the stage of processing the aggregate system of equations (see **Section 2.6**). An example of an underdetermined system of equations (x, y, and z are free variables):

$$x + y + z = 1$$
$$x - y = 0$$

The number of free variables cannot be less than the number of unknown variables that Rand Model Designer needs to select. If the number of free variable is less than the required number of unknown variables, it indicates that the system of equations is *overdetermined*. An overdetermined system of equations is either redundant or non-resolvable. Rand Model Designer, therefore, disallows overdetermined systems of equations. An example of an overdetermined system of equations:

$$x + y = 1$$
$$x - y = 0$$
$$x - 2 \cdot y = 0$$

Another example:

$$\frac{dx}{dt} + x = 1$$

$$\frac{dx}{dt} + x = 1$$

The system of equations of a class has the following effect on the model: during the model execution the system of equations of each instance of the class is automatically solved as part of the aggregate system of equations (see **Section 2.6**), thereby realizing the behavior of each instance. The system of equations of each instance is solved starting from whatever the initial variable values are in that instance.

If the system of equations of a class includes second-order differential equations, it must also specify the initial values of the first derivatives of the corresponding variables. The initial value of the first derivative of a variable can be specified using the following syntax:

```
<variable name>'(0) = <initial value>
```

For example:

```
x'(0) = 1
```

If the system of equations of a class contains a set of equations whose variables do not participate in other equations of the system and are internal variables of the class (see **Section 2.9**), such a set of equations can be declared as a *block*. A block declaration has the following syntax:

```
block <block name>
<equation 1>
...
end block
```

For example:

```
block myblock
```

$$\frac{dx}{dt} = y$$

$$\frac{dy}{dt} = -x$$

```
end block
```

Declaring a set of equations as a block gives Rand Model Designer a hint that the equations of the block can be solved separately from other equations of the system. Rand Model Designer may take advantage of such hints when solving the aggregate system of equations.

2.6 AGGREGATE SYSTEM OF EQUATIONS

An *aggregate system of equations* is a join of the systems of equations of all object instances existing in the executable model at a particular time moment. When executing a model, Rand Model Designer solves the aggregate system of equations numerically, thereby realizing the continuous behavior of the model.

Depending on the model, the aggregate system of equations may remain constant or may change during the model execution. The aggregate system of equations changes every time some continuous object appears in or disappears from the

model. A continuous object may appear in or disappear from the model as a result of transition of some hybrid object from one state to another (see **Section 2.4**). This mechanism allows models to exhibit hybrid behavior.

The selection of unknown variables for which the aggregate system of equations is to be solved is performed by Rand Model Designer at the model start and every time the aggregate system of equations changes. Variables selected as known and unknown in the systems of equations of classes become known and unknown, respectively, in the aggregate system of equations. If the resulting number of unknown variables turns out to be equal to the number of equations in the aggregate system, the aggregate system is considered formally resolvable and is solved in its existing form. If the number of unknown variables turns out to be less than the number of equations in the aggregate system, Rand Model Designer adjusts the aggregate system by arbitrarily selecting the required number of unknown variables from the set of free variables and declaring the remaining free variables known. The resulting system is considered formally resolvable and is solved in its adjusted form.

2.7 STRUCTURE DIAGRAM

If a class is compound and/or open (see **Section 2.3**), it has the associated *structure diagram*. If a class is compound, its structure diagram specifies the internal structure of the class. If a class is open, its structure diagram defines the interface of the class. If a class is compound and open at the same time, its structure diagram serves both functions.

A structure diagram can contain:

Local objects

Links

The class boundary

Local objects define the internal structure of a compound class. Local objects are the elements from which a compound class is built.

Links define how local objects interact with each other and the class itself. Links are connections between variables.

The *class boundary* defines the interface of an open class.

2.7.1 Local Object

If a class is compound, its structure diagram may contain and usually does contain one or more *local objects*. A local object is shown on a structure diagram as a rectangle, see **Figure 12**.

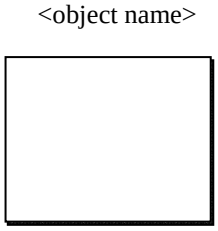


Figure 12 Local object

A local object has the following properties:

Name

Class

Actual parameter values

Initial variable values

The *name* of a local object is an identifier (see **Section 6.2**) unique across all local objects of the structure diagram.

The *class* of a local object can be any class available in the project (see **Section 2.2**).

The presence of a local object on the structure diagram of a class indicates that each instance of the class contains an instance of the local object's class, with the specified name, actual parameter values (see **Section 2.8**) and initial variable values (see **Section 2.9**).

If the class of a local object declares external variables (see **Section 2.9**), the corresponding external variables of the local object appear on the border of the local object as shown in **Figure 13**. The appearance of a particular variable depends on the category of the variable. **Figure 13** depicts the appearance of variables of each possible category. For the discussion of variable categories, see **Section 2.9**.

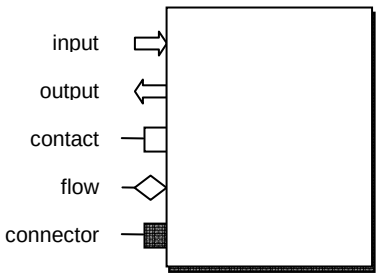


Figure 13 Appearance of external variables

2.7.2 Class Boundary

If a class is open, its structure diagram contains the *class boundary*. A class boundary is shown on a structure diagram as a rectangle, see **Figure 14**.

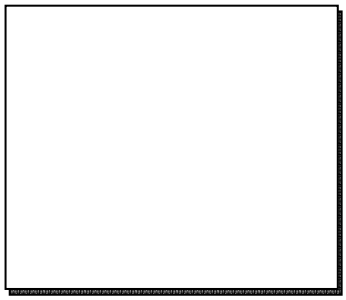


Figure 14 Class boundary

If a class has external variables (see **Section 2.9**), they appear on the class boundary in the same way as external variables of local objects (see **Section 2.7.1**).

2.7.3 Link

A *link* is a connection between two variables on a structure diagram. A link is shown as a polyline consisting of horizontal and vertical segments, see **Figure 15**.

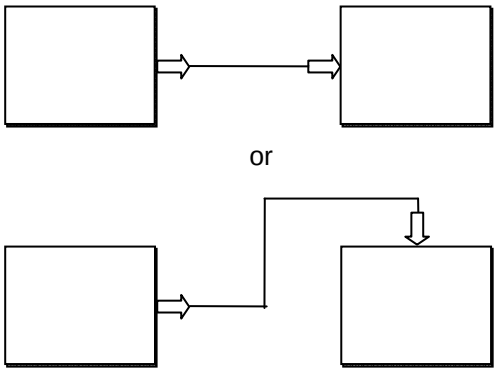


Figure 15 Link examples

A link can connect:

An output variable of one local object with an input variable of another or same local object, see **Figure 16**

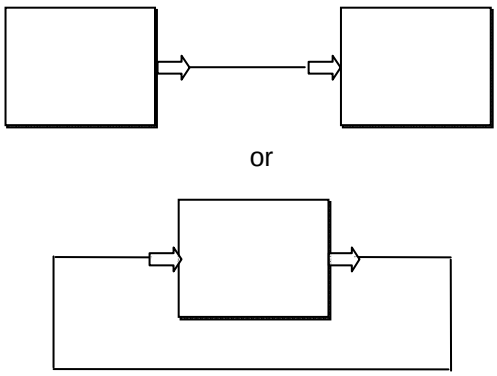


Figure 16 Output-to-input connection

An input variable of the class with an input variable of a local object, see **Figure 17**

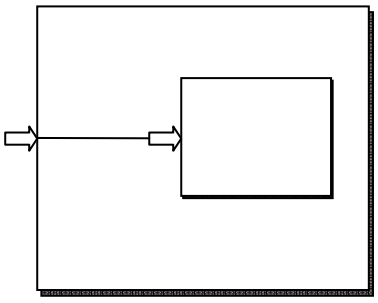


Figure 17 Input-to-input connection

An output variable of a local object with an output variable of the class, see **Figure 18**

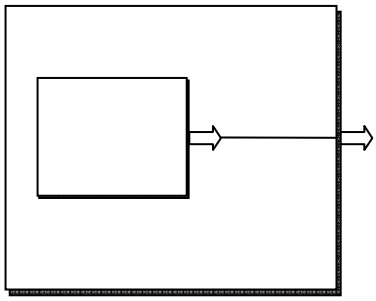
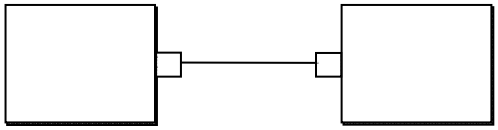
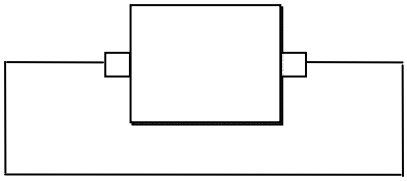


Figure 18 Output-to-output connection

A contact variable of one local object with a contact variable of another or same local object or a contact variable of the class, see **Figure 19**



or



or

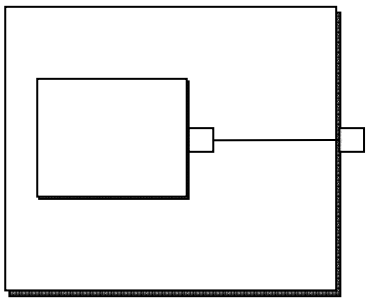
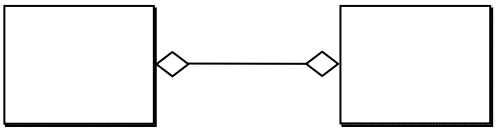
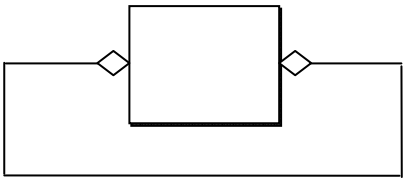


Figure 19 Contact-to-contact connection

A flow variable of one local object with a flow variable of another or same local object or a flow variable of the class, see **Figure 20**



or



or

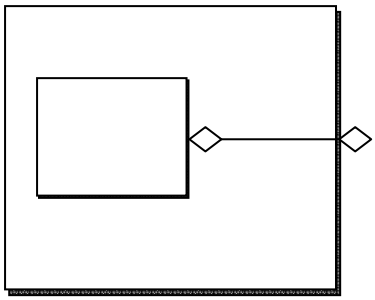
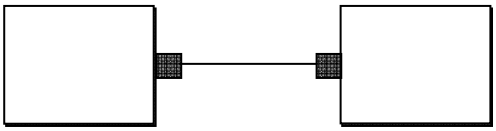
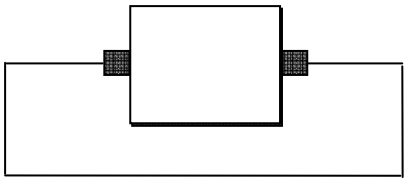


Figure 20 Flow-to-flow connection

A connector variable of one local object with a connector variable of another or same local object or a connector variable of the class, see **Figure 21**



or



or

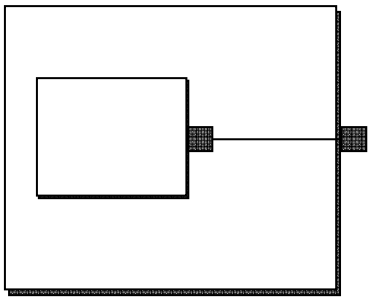


Figure 21 Connector-to-connector connection

A variable may not be connected to itself. Every pair of variables may be connected by only one link. An input variable of a local object may be connect to only one other variable. An output variable of the class may also be connected to only one other variable. Any other variable may be connected to any number of other variables.

A link between two connector variables connects the components of these variables pairwise in the order of declaration of components in the types of these variables (see **Section 2.13.4**). Two connector variables may only be connected if each respective pair of their components is connectable.

Every link supplements the system of equations of the class (see **Section 2.5**) as follows.

If `obj1.out` is an output variable of one local object and `obj2.in` is an input variable of another or same local object, a link between `obj1.out` and `obj2.in` adds the following to the system of equations of the class:

```
obj2.in = obj1.out
known obj1.out
unknown obj2.in
```

If `in` is an input variable of the class and `obj.in` is an input variable of a local object, a link between `in` and `obj.in` adds the following to the system of equations of the class:

```
obj.in = in
known in
unknown obj.in
```

If `obj.out` is an output variable of a local object and `out` is an output variable of the class, a link between `obj.out` and `out` adds the following to the system of equations of the class:

```
out = obj.out
known obj.out
unknown out
```

If `x` and `y` are contact variables, a link between `x` and `y` adds the following equation to the system of equations of the class:

```
x = y
```

If `x` and `y` are flow variables, a link between `x` and `y` adds the following equation to the system of equations of the class:

```
x + y = 0
```

If `x` and `y` are connector variables, a link between them is equivalent to a set of links connecting their components in the order of declaration of components in the types of these variables (see **Section 2.13.4**).

2.8 PARAMETERS

A class can have an arbitrary number of *parameters*. Every parameter has the following properties:

Name

Type

Default value

The *name* of a parameter is an identifier (see **Section 6.2**) unique across all parameters of the class.

The *type* of a parameter can be any available type except for signal and connector types (see **Section 2.13**).

The *default value* is specified by an expression (see **Section 2.15**) that may reference other parameters of the class, however, such references are not allowed to form cycles.

A parameter definition within a class designates a *formal parameter*. A formal parameter of a class receives its actual value in each instance of the class. The actual value can be assigned to a parameter via the set of actual parameter values of the corresponding local object (see **Section 2.7.1**) or activity object (see **Section 2.4.1**). If the actual value of a parameter is not specified, the default value specified in the parameter definition is used.

Every parameter is assigned its actual value once upon creation of the corresponding instance of the class. Once the value is assigned, it cannot be changed.

2.9 VARIABLES

A class can have an arbitrary number of *variables*. Variables store the data of the class and may be used by the class to interact with its environment.

If a class is open (see **Section 2.3**), it can have variables of two kinds: *external* and *internal*. External variables define the interface of the class. If a class has external

variables, they appear on the class boundary on the structure diagram of the class (see **Section 2.7.2**). Also, if some structure diagram contains a local object whose class has external variables, those external variables appear on the border of the local object (see **Section 2.7.1**). Internal variables belong to the internal implementation of the class and have no direct effect on the class interface.

If a class is isolated, it can only have internal variables. Internal variables of an isolated class are usually called simply variables.

Every variable has the following properties:

Name

Category (if the variable is external)

Type

Default initial value

The *name* of a variable is an identifier (see **Section 6.2**) unique across all variables of the class.

If a variable is external, it has a *category*. The category defines the role of the variable in the class interface. Possible variable categories are:

Input

Output

Contact

Flow

Connector

An *input* variable can transfer information in only one direction, namely, to the class. The value of an input variable can only be changed from outside the class.

An *output* variable can transfer information in only one direction, namely, from the class. The value of an output variable can only be changed from inside the class.

The value of a *contact* variable can be changed from both inside and outside the class. The values of any two contact variables connected with a link are always equal.

The value of a *flow* variable can be changed from both inside and outside the class. The sum of values of flow variables connected with links is always equal to `0`.

A *connector* variable is a set of one or more components each of which is equivalent to an individual external variable and can, in its turn, be of any category (see **Section 2.13.4**). Connector variables are useful for modeling physical systems of various kinds including electric circuits.

The *type* of a variable specifies the set of values that can be stored in that variable (see **Section 2.13**). A contact or flow variable can only be of type `double` (see **Section 2.13.1.1**). A connector variable can only be of a connector type (see **Section 2.13.4**). Any other variable can be of any available type except for connector types.

The *default initial value* of a variable is specified by an expression (see **Section 2.15**) that may reference other variables of the class, however, such references are not allowed to form cycles.

A variable has some value in each instance of the class. A variable is assigned its initial value upon creation of the corresponding instance. After that, the variable

value can change in response to explicit assignments or as the result of solving the aggregate system of equations (see **Section 2.6**).

The initial value can be assigned to a variable via the set of initial variable values of the corresponding local object (see **Section 2.7.1**) or activity object (see **Section 2.4.1**). If the initial value of a variable is not specified, the default initial value specified in the variable definition is used.

Links between variables are discussed in **Section 2.7.3**.

2.10 CONSTANTS

Both a class (see **Section 2.2**) and a project (see **Section 2.1**) can define an arbitrary number of *constants*. A constant is a fixed value that has an associated symbolic name.

Every constant has the following properties:

Name

Type

Value

Visibility (if the constant is a project constant)

The *name* of a constant is an identifier (see **Section 6.2**) unique across all constants of the class or project.

The *type* of a constant can be any available type except for signal and connector types (see **Section 2.13**).

The *value* of a constant is specified by an expression (see **Section 2.15**). The expression specifying the value of a class constant may reference other constants of the class and any constants of the project. The expression specifying the value of a project constant may reference other constants of the project. In either case, such references are not allowed to form cycles.

If a constant is defined in a project, it has *visibility*. The visibility indicates that the project constant is either *private* or *public*. A private constant can only be used in the project containing that constant. A public constant can be used in any project. For the detailed discussion of the visibility of project components, see **Section 2.1**.

Class constants differ from project constants in their scope, see **Section 2.17**. Project constants are visible in the entire project. Moreover, public project constants are visible in all other projects that import their containing project. Class constants are only visible within their containing class.

2.11 LOCAL CLASSES

A hybrid class can contain an arbitrary number of *local classes*. A local class is a class defined in a hybrid class. Local classes can be used to associate activities with behavior chart states (see **Section 2.4**) of their containing class.

All in all, local classes have the same semantics as ordinary classes (see **Section 2.2**), except for the following differences:

A local class is an element of some other class, whereas an ordinary class is an element of a project.

A local class is only visible in its containing class. A local class is not visible in other classes of the same project or other projects.

A local class may not have no behavior.

All parameters, variables, constants, functions, and procedures of the containing class are automatically visible in local classes. This makes local classes convenient for implementing state activities.

A local class may not be abstract and may not have subclasses.

2.12 CLASS INHERITANCE

Rand Model Designer supports *class inheritance*. Class inheritance allows the user to create new classes from existing ones.

A class may have a *base class*. A class that has a base class is called a *derived class*. A derived class *inherits* all elements and characteristics of the base class, including the stereotype, behavior chart, system of equations, structure diagram, parameters, variables, constants, functions, and procedures. If necessary, a derived class may *extend* the base class. A derived class may:

Widen the class stereotype. For example, if the base class is isolated, the derived class can be open.

Add new equations to the inherited system of equations

Override the inherited equations. Overriding any subset of the inherited equations requires overriding the entire system of equations.

Add new states and transitions to the inherited behavior chart

Override the activity associated with an inherited state, including the actual parameter values of the activity object

Override the entry and exist actions of an inherited state

Override the trigger, the guard, and the actions of an inherited transition

Add new local objects and links to the inherited structure diagram

Override the actual parameter values of an inherited local object

In a derived class, all inherited elements are labeled with the ⬆ sign. If a derived class overrides the system of equations, the overridden system of equations is labeled with the ♦ sign. The inherited and overridden states, transitions, local objects, and links are shown on the behavior chart and the structure diagram in special colors configurable by the user.

None of the elements of the base class can be removed in the derived class.

In Rand Model Design, a base class is often called a *superclass*, and a derived class is often called a *subclass*.

2.13 TYPES

A *type* defines a set of values that an object or expression can take and a set of operations allowed on those values. Every type has a *name* that uniquely identifies the type. The use of a particular type name in a definition of an object, such as parameter, variable, or constant, indicates that the corresponding object has the corresponding type, that is, can take values from the corresponding set.

2.13.1 Scalar Types

The scalar types are:

`double`

`byte`

`short`

`integer`

`boolean`

`char`

`string`

user-defined enumerations

2.13.1.1 double

The `double` type represents a double precision 64-bit floating-point number with values ranging from `4.9E-324` to `1.7E+308`. The `double` type and associated operations generally adhere to the IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985). The `double` type is used to approximately represent real numbers.

The operations defined for the `double` type are (see also **Section 2.15.1**):

”+” – addition

"`-`" – subtraction

"`*`" – multiplication

"`/`" – division

"`**`" – raising a number to the power of another number

"=" – equality relation

"`!=`" – inequality relation

">" – “greater than” relation

">=" – “greater than or equal to” relation

"<" – “less than” relation

"<=" – “less than or equal to” relation

Literals of the `double` type have the following syntax (here and later in this document, the Extended Backus-Naur Form is employed to describe the syntax of certain language constructs, see **Section 6.1**):

```
<double_literal> ::= [<sign>]<significand>[<exponent>]

<sign> :: = +!-

<significand> ::=
  <digits>.[<digits>]

<digits> ::=
  <digit>{<digit>}

<digit> ::= 0!1!2!3!4!5!6!7!8!9

<exponent> ::= <exponent_symbol>[<sign>]<digits>
```

```
<exponent_symbol> ::= E!e
```

Examples:

```
-3.5
+5.67
1.5e3
-3.4E12
1.76E-2
```

2.13.1.2 byte, short, integer

The `byte`, `short`, and `integer` types represent integer numbers.

The `byte` type represents an 8-bit unsigned integer number ranging from `0` to `255`.

The `short` type represents a 16-bit signed integer number ranging from `-32768` to `32767`.

The `integer` type represents a 32-bit signed integer number ranging from `-2147483648` to `2147483647`.

The operations defined for the `byte`, `short`, and `integer` types are (see also **Section 2.15.1**):

"+" – addition

"`-`" – subtraction

"`*`" – multiplication

"`div`" – integer division

"`mod`" – remainder calculation (modulo operator)

"`**`" – raising a number to the power of another number

"`or`" – bitwise “or”

"`and`" – bitwise “and”

"`not`" – bitwise “not”

"=" – equality relation

"`!=`" – inequality relation

">" – “greater than” relation

">=" – “greater than or equal to” relation

"<" – “less than” relation

"<=" – “less than or equal to” relation

`byte`, `short`, and `integer` literals have the following syntax:

```
<integer_literal> ::= [<sign>]<digits>

<sign> ::= +!-

<digits> ::=
  <digit>{<digit>}

<digit> ::= 0!1!2!3!4!5!6!7!8!9
```

Examples:

```
134

-4567
```

2.13.1.3 boolean

The `boolean` type represents a logical quantity with two possible values: `true` and `false`.

The operations defined for the `boolean` type are (see also **Section 2.15.1**):

"`or`" – logical “or”

"`and`" – logical “and”

"`not`" – logical “not”

"=" – equality relation

"`!=`" – inequality relation

The syntax of `boolean` literals:

```
<boolean_literal> ::= false!true
```

2.13.1.4 char

The `char` type represents a single text character.

The operations defined for the `char` type (see also **Section 2.15.1**):

"=" – equality relation

"`!=`" – inequality relation

`char` literals are single characters enclosed in double quotes, for example:

```
"A"

"1"

"a"
```

2.13.1.5 string

The `string` type represents a variable length string of characters.

The operations defined for the `string` type (see also **Section 2.15.1**):

"+" – concatenation

"=" – equality relation

"`!=`" – inequality relation

`string` literals are character strings enclosed in double quotes, for example:

```
"AbCD"

"1234"

"a1B7"
```

2.13.1.6 Enumerations

An *enumeration* is a type whose set of values – an ordered sequence of literals – is specified by the user. Every literal of an enumeration is an identifier (see **Section 6.2**) unique across all literals of the enumeration.

An example of a sequence of literals specifying an enumeration:

```
North, East, South, West
```

The operations defined for every enumeration are (see also **Section 2.15.1**):

"=" – equality relation

"!=" – inequality relation

">" – “greater than” relation

">=" – “greater than or equal to” relation

"<" – “less than” relation

"<=" – “less than or equal to” relation

Two enumeration types are identical if their sets of literals are identical.

2.13.2 Regular Types

The regular types are fixed- and variable-size *vectors* and *matrices*.

2.13.2.1 Vectors

A `vector[N]` type represents a fixed-size vector with `N` elements of type `double`. The `vector` type represents a variable-size vector with elements of type `double`. Vector elements are numbered starting from `1`. The current size of a vector can be obtained using the `size` function (see **Section 2.14.1.2**).

The operations defined for vectors are (see also **Section 2.15.1**):

"*" – multiplication by a scalar

"+" – addition

"-" – subtraction

"=" – equality relation

"!=" – inequality relation

Vector literals have the following syntax:

```
<vector_literal> ::=
  "["<double_literal>{;<double_literal>}"]"
```

Examples:

```
[1; 2; 3; 4]
```

```
[0; 0; 2.3; 5.67; 1E2]
```

2.13.2.2 Matrices

A `matrix[N,M]` type represents a fixed-size matrix with `N` rows and `M` columns with elements of type `double`. The `matrix` type represents a variable-size matrix with elements of type `double`. Row and columns are numbered starting from `1`. Every vector is and can be used as a matrix with one column – `matrix[N,1]`. The current size of a matrix can be obtained using the `size` function (see **Section 2.14.1.2**).

The operations defined for matrices are (see also **Section 2.15.1**):

"*" – multiplication by a scalar or another matrix

"+" – addition

"-" – subtraction

"=" – equality relation

"!=" – inequality relation

Matrix literals have the following syntax:

```
<matrix_literal> ::=
  "["<matrix_row>{;<matrix_row>}"]"
  !<itertive_matrix_literal>

<matrix_row> ::=
  <double_literal>{,<double_literal>}

<itertive_matrix_literal> ::=
  "["<iteration_spec>|<double_expression>"]"

<iteration_spec>::=
  for <row_iteration_variable> in <range>
  [,<column_iteration_variable> in <range>]

<row_iteration_variable> ::= <identifier>

<column_iteration_variable> ::= <identifier>

<range> ::= <lower_bound>..<upper_bound>
```

where <double_expression> is an expression (see **Section 2.15**) that evaluates to double, <lower_bound> and <upper_bound> are static expressions that evaluate to integer, and <identifier> is an identifier (see **Section 6.2**).

Matrix literal examples:

```
[ 2, 3.5; 1, 10 ]

[ for i in 1..3, j in 1..3 | i*j ]

[ for i in 1..size(A,1), j in 1..size(A,2) | A[i,j]**2 ]
```

In an iterative matrix literal, the lower bound of an iteration range may not be less than 1. If the lower bound of an iteration range is greater than 1, the skipped rows and/or columns are filled with zeros.

2.13.3 Record

A *record* is a type that represents a collection of one or more components each of which has its own *name* and *type*. Record components are often called *fields*.

The name of a record field is an identifier (see **Section 6.2**) unique across all fields of the record.

The *type* of a record field can be any available type except for signal and connector types.

An MVL example (see **Section 6.3**) declaring a record type:

```
type MyRecord is
  record
    C1:double;
    C2:string;
    C3:vector[3];
  end record;
```

2.13.4 Connector

A *connector* is a type applicable only to external variables of a class (see **Section 2.9**). A connector type represents a collection of one or more *components* each of which is analogous to an individual external variable and has the following properties:

Name

Category

Type

The *name* of a connector component is an identifier (see **Section 6.2**) unique across all components of the connector.

The *category* of a connector component has the same meaning and possible values as the category of an external variable.

The *type* of a connector component can be any available type.

A connector variable can be viewed as a “socket” that can be connected to other compatible “sockets”; the components of a connector variable can be viewed as the pins of the “socket”.

A link between two connector variables connects the components of these variables pairwise in the order of declaration of components in the types of these variables. Two connector variables may only be connected if each respective pair of their components is connectable (see **Section 2.7.3**).

An MVL example (see **Section 6.3**) declaring a connector type:

```
type MyConnector is
  connector
    contact U:double;
    flow I:double;
  end connector;
```

2.13.5 Signal

A *signal* is a type applicable only to input, output, and internal variables of a class (see **Section 2.9**). A variable of a signal type is able to transmit signals – discrete events that can trigger behavior chart transitions (see **Section 2.4.2**).

Signals can carry additional information in their *parameters*. Signal parameters are similar to record fields.

A signal type represents a collection of one or more parameters each of which has its own *name* and *type*.

The name of a parameter is an identifier (see **Section 6.2**) unique across all parameters of the signal type.

The *type* of a parameter can be any available type except for signal and connector types.

An MVL example (see **Section 6.3**) declaring a signal type:

```
type MySignal is
  signal (P1:double; P2:matrix; P3:(a,v,k));
```

A signal can be created and sent to a variable of the appropriate type using the `send` operator (see **Section 2.16.8**).

The parameter values of a signal can only be accessed by the actions of the transitions triggered by that signal. In the actions of a transition triggered by a signal a particular parameter of the signal can be accessed as:

```
<variable name>.<parameter name>
```

where `<variable name>` is the name of the variable that received the signal and `<parameter name>` is the name of the parameter.

2.13.6 Declared Types

A *declared type* is a project component that acts as an alias, or synonym, of some existing type. Once created, a declared type can be used as any other type.

A project can contain an arbitrary number of declared types (see **Section 2.1**).

The definition of a declared type consists of the *name* of the type and the *specification* of the type.

The name of a declared type is an identifier (see **Section 6.2**) unique across all declared types of the project.

The specification of a declared type defines the type for which the declared type will act as a synonym.

Consider an MVL example (see **Section 6.3**):

```
type Alphabet is (a,b,c);
```

This example creates a declared type `Alphabet` that acts as a synonym for the `(a,b,c)` enumeration. The new type can be used, for example, as a variable type:

```
var:Alphabet := a;
```

This is equivalent to using the same enumeration type directly:

```
var:(a,b,c) := a;
```

Record and connector types cannot be used as types of parameters, variables, constants, etc directly and can only play the role of the specification in declared types. This means that in order to define, for example, a variable of a record or connector type, the user should first declare a type representing the required record or connector type, and then define a variable of the declared type. Other types can be used as types of parameters, variables, constants, etc both directly and through declared types.

Declared types have public visibility. This means that every declared type can be used in both its containing project and any other project. For the detailed discussion of the visibility of project components, see **Section 2.1**.

2.14 FUNCTIONS AND PROCEDURES

Functions and procedures are subroutines that can be called from various places within a project. A function is a subroutine that returns some value. A procedure is a subroutine that does not return a value.

A function call executes the body of the called function and receives the value returned by the function. The returned value can participate in the evaluation of the expression that called the function. The syntax of function calls is described in **Section 2.15**.

A procedure call executes the body of the called procedure and does not receive any value. The syntax of procedure calls is described in **Section 2.16.2**.

Functions and procedures can be divided into *predefined functions and procedures* and *user-defined functions and procedures*. The predefined functions and procedures are defined and automatically available in all projects. User-defined functions and procedures become available only after they are explicitly defined by the user.

The predefined functions and procedures are described in the section that follows. User-defined functions and procedures are discussed in **Section 2.14.2**.

2.14.1 Predefined Functions and Procedures

The following predefined functions and procedures are available in all projects.

2.14.1.1 Standard Math Functions

`abs(x)` – returns the absolute value of `x`.

`arccos(x)` – returns the arccosine of `x`, requires $-1 \le x \le 1$.

`arch(x)` – returns the inverse hyperbolic cosine of `x`.

`arcsin(x)` – returns the arcsine of `x`, requires $-1 \le x \le 1$.

`arctg(x)` – returns the arctangent of `x`.

`arsh(x)` – returns the inverse hyperbolic sine of `x`.

`arth(x)` – returns the inverse hyperbolic tangent of `x`.

`ch(x)` – returns the hyperbolic cosine of `x`.

`cos(x)` – returns the cosine of `x`, where `x` is specified in radians.

`ctg(x)` – returns the cotangent of `x`, where `x` is specified in radians. Requires $x \ne \pm\pi \cdot n$.

`deg(x)` – converts `x` from radians to degrees.

`exp(x)` – returns the exponential of `x`.

`lg(x)` – returns the base-10 logarithm of `x`, requires $x > 0$.

`ln(x)` – returns the natural logarithm of `x`, requires $x > 0$.

`max(x1, x2, …, xn)` – returns the maximum among the arguments.

`min(x1, x2, …, xn)` – returns the minimum among the arguments.

`rad(x)` – converts `x` from degrees to radians.

`round(x)` – rounds `x` to the nearest integer.

`sh(x)` – returns the hyperbolic sine of `x`.

`sign(x)` – returns `1` if $x \ge 0$, `-1` if $x < 0$.

`sin(x)` – returns the sine of `x`, where `x` is specified in radians.

`sqrt(x)` – returns the square root of `x`, requires $x \ge 0$.

`tg(x)` – returns the tangent of `x`, where `x` is specified in radians. Requires $x \ne \frac{\pi}{2} \pm \pi \cdot n$.

`th(x)` – returns the hyperbolic tangent of `x`.

`trunc(x)` – rounds `x` towards zero to the nearest integer, that is, returns the integer part of `x`.

2.14.1.2 Functions and Procedures for Vectors and Matrices

`column(x,j)` – returns the `j`-th column of the matrix `x`.

`det(x)` – returns the determinant of the square matrix `x`.

`eye(n)` – returns an n-by-n square matrix with ones on the diagonal and zeros elsewhere.

`inv(x)` – returns the inverse of the square matrix `x`.

`matrixfromfile(fp)` – loads a matrix from a text file. `fp` specifies the path to the file. If `fp` is a relative path, it is assumed to be relative to the project folder. The first line of the file is assumed to contain column names and, therefore, is ignored. If different lines of the file have different number of elements, the number of columns will be equal to the maximum number of elements and all unspecified elements will be set to `0`.

`matrixtofile(x,fp)` – stores the matrix or vector `x` as a table into a text file. `fp` specifies the path to the file. If `fp` is a relative path, it is assumed to be relative to the project folder.

`max(x)` – returns the largest element of `x`, where `x` is either a vector or matrix.

`mean(x)` – returns the mean value among all elements of `x`, where `x` is either a vector or matrix.

`min(x)` – returns the smallest element of `x`, where `x` is either a vector or matrix.

`norm1(x)` – returns the 1-norm of the matrix `x`.

`normE(x)` – returns the Euclidean norm of the matrix `x`.

`normInf(x)` – returns the infinity-norm of the matrix `x`.

`normspectr(x)` – returns the spectral norm (2-norm) of the matrix `x`.

`norm_m(x)` – returns the M-norm of the matrix `x`.

`ones(n)` – returns a vector of size `n` filled with ones.

`ones(n,m)` – returns an n-by-m matrix of ones.

`putcolumn(x,j,c)` – returns the matrix `x` with the `j`-th column replaced with the vector `c`.

`putrow(x,i,r)` – returns the matrix `x` with `i`-th row replaced with the row matrix (1-by-m matrix) `r`.

`row(x,i)` – returns the `i`-th row of the matrix `x`.

`scalar(x,y)` – returns the scalar product of the vectors `x` and `y`.

`size(x)` – returns the size of the vector `x`. Returns `0` if `x` is a variable-size vector that has not been assigned any value yet.

`size(x,i)` – returns the size of the `i`-th dimension of the matrix `x`. Returns `0` if `x` is a variable-size matrix that has not been assigned any value yet.

`submatrix(x,i,j,n,m)` – returns an n-by-`m` sub-matrix of the matrix `x` starting from `x[i,j]` as the top-left element.

`sum(x)` – returns the sum of all elements of `x`, where `x` is either a vector or a matrix.

`transp(x)` – returns the transpose of the matrix `x`.

`variance(x)` – returns the variance of the vector `x`.

`vectorfromfile(fp,cn)`, `vectorfromfile(fp,cj)` – loads a vector from a text file. `fp` specifies the path to the file. If `fp` is a relative path, it is assumed to be relative to the project folder. `cj` specifies the index of the column to be loaded as a vector, for example `vectorfromfile("file.txt",2)`. `cn` specifies the name of the column to be loaded as a vector, for example `vectorfromfile("file.txt","A")`. The file is assumed to contain the column names in the first line.

`vnorm1(x)` – returns the 1-norm of the vector `x`.

`vnorm2(x)` – returns the Euclidean norm of the vector `x`.

`vnormInf(x)` – returns the infinity-norm of the vector `x`.

`zeros(n)` – returns a vector of size n filled with zeros.

`zeros(n,m)` – returns an n-by-m matrix of zeros.

2.14.1.3 Random Number Generators

`expon(m)` – generates a random number from the exponential distribution with mean parameter `m`.

`normal(m,d)` – generates a random number from the normal distribution with mean parameter `m` and variance parameter `d`.

`uniform(xmin,xmax)` – generates a random number from the uniform distribution on the interval `[xmin; xmax]`.

2.14.1.4 Special Functions and Procedures

`BreakDiagrams` – inserts a break into all time and phase diagrams opened in the visual executable model causing the next point to be disconnected from the previous point in all plots. Ignored in the embeddable model.

`ClearDiagrams` – clears all time and phase diagrams opened in the visual executable model. Ignored in the embeddable model.

`delay(x,t)` – returns the value of the variable `x` at time moment `time - t`, where `time` is the current model time of the calling object. During the time interval `0 ≤ time ≤ t`, returns the initial value of `x`.

`deriv(x)` – returns the value of the derivative of `x` for the current time moment, where `x` is a variable of type `double` or `vector`. The value of the derivative is calculated numerically, so at the time moment `time = 0` the return value is always `0` for `double` variables and `[0;0;...]` for `vector` variables (here `time` is the current model time of the calling object).

`Error(m)` – opens a message box with the message `m` and stops the model execution. Displays the name of the calling object in the title of the message box.

`Finalized(x)` – returns `true` if the object `x` is hybrid and has already reached its final state (see **Section 2.4.5**), returns `false` otherwise.

`getFilePath(title, fexts)` – opens a dialog box that allows the user to select a file. Returns the path to the selected file. `title` specifies the dialog box title. `fexts` specifies the comma-separated list of allowed file extensions. Example: `getFilePath("Select file","txt,tab")`.

`Infinity` – returns the “infinity” value as defined by ANSI/IEEE Std 754-1985.

`InputVar(x)` – opens a dialog box that allows the user to modify the value of the variable `x`. Ignored in the embeddable model.

`InstanceHandle` – returns the integer instance ID of this object. Instance IDs are unique across all instances of a particular class, but are not unique across different classes.

`interpol(fx,fy,x)` – returns an interpolated value of the function whose values `fy` are tabulated for points `fx`. `x` specifies the argument for which to return the interpolated value. Returns `NaN` if `x` is outside the interval spanned by `fx`.

`interpolXY(xy,x)` – returns an interpolated value of the function specified by the n-by-2 matrix `xy` whose first column contains the argument values and the second column contains the corresponding function values. `x` specifies the argument for which to return the interpolated value. Returns `NaN` if `x` is outside the interval spanned by the values found in the first column of `xy`.

`Interim(mode)` – if called with the `mode` parameter set to `true`, causes Rand Model Designer to stop analyzing and solving the aggregate system of equations until the end of the current time gap or a call to this procedure with the `mode` parameter set to `false`.

`IsInfinite(x)` – returns `true` if `x` is an “infinity”, `false` otherwise.

`IsNaN(x)` – returns `true` if `x` is a “not a number” (`NaN`), `false` otherwise.

`length(x)` – returns the number of characters in the string `x`.

`MainWinHandle` – returns a handle to the main window of the visual executable model. The returned handle can be used to, for example, open additional windows from within the visual executable model.

`Message(t,m)` – opens a message box with the title `t` and message `m` and pauses the model execution until the message box is closed. Ignored in the embeddable model.

`ModelTime` – returns the current model time, that is, the amount of model time elapsed since the start of the model, in model time units.

`NaN` – returns a “not a number” (`NaN`) value as defined by ANSI/IEEE Std 754-1985.

`ObjectHandle` – returns the unique integer ID of the class of this object.

`OnVarChange(x)` – returns `true` if the variable `x` was modified during the current discrete time tick. Returns `false` otherwise.

`StateResult` – may only be used in the exit actions of a state (see **Section 2.4.1**). Returns `true` if there were no errors while being in the state, `false` otherwise.

`Stop` – causes Rand Model Designer to stop the model execution after finishing executing the transition whose actions called this procedure (see **Section 2.4.2**).

`time` – returns the current model time of this object, that is, the amount of model time elapsed since the creation of this object, in model time units.

`Time` – if this object is an activity object, returns the current model time of the object owning the behavior chart with the corresponding state. Otherwise returns the current model time of the object.

`tostring(x)` – returns a string representation of `x`.

2.14.1.5 Functors

A *functor* is a function that accepts an expression, rather than a value, as one of its arguments. A functor call looks like an ordinary function call but is in fact a special operator that specifies an independent variable, an expression involving the independent variable, an interval for the independent variable, and, optionally, an absolute error tolerance for the result. A functor call has the following syntax:

```
<functor_call> ::=
  <functor_name>(
    <independent_variable_name>,
    <expression>,
    <independent_variable_lower_bound>,
    <independent_variable_upper_bound >[,
    <absolute_tolerance>]
  )
```

The independent variable must be of type `double`. The expression must evaluate to `double`. The scope of the independent variable is limited to the expression. Besides the independent variable, the expression can involve any other quantities such as class variables, parameters, and constants, but during the calculation of the functor all quantities remain constant except for the independent variable. If the absolute error tolerance is omitted, the default specified in the numerical method settings is used.

The following functors are supported:

2.14.1.5.1 integral

`integral` calculates the definite integral of the specified function on the specified interval.

Example:

```
integral(x, k*sin(x), 0, 1)
```

In this example, `x` is the independent variable, `k*sin(x)` is the expression specifying the function to be integrated, 0 and 1 are the lower and upper bounds of the integration interval respectively. `k` can be a variable, a parameter, or any other quantity visible at this point.

The lower and upper bounds can be infinite, for example:

```
integral(x, exp(-sign(x)*x), -Infinity, Infinity)
```

2.14.1.5.2 maxfun

`maxfun` finds and returns the independent variable value that corresponds to the maximum value of the specified function on the specified interval.

Example:

```
maxfun(z, z**2+2, 1, 2)
```

2.14.1.5.3 minfun

`minfun` finds and returns the independent variable value that corresponds to the maximum value of the specified function on the specified interval.

Example:

```
minfun(z, z**2+2, 1, 2)
```

2.14.1.5.4 root

`root` finds and returns one root of the algebraic equation of the form `<expression>=0`. For example, in order to find the root of the equation `x**2–2*x–3=0` on the interval `[0;10]`, the `root` functor can be called as follows:

```
root(x, x**2–2*x-3, 0, 10)
```

If the specified equation has zero or more than one root on the specified interval, an interrupt is raised.

2.14.2 User-Defined Functions and Procedures

Every class (see **Section 2.2**) and every project (see **Section 2.1**) can contain an arbitrary number of *user-defined functions and procedures*. User-defined functions and procedures are subroutines that are created by the user.

A user-defined function or procedure can be implemented either in the programming language offered by Rand Model Designer or in any other programming language outside Rand Model Designer with the use of third party tools such as Visual C++, Delphi, Fortran, etc. Functions and procedures implemented in the Rand Model Designer programming language are called *internal*. Functions and procedures implemented outside Rand Model Designer are called *external*.

If a function or procedure is defined in a project, it has *visibility*. The visibility indicates that the project function or procedure is either *private* or *public*. A private function or procedure can only be used in the project containing that function or procedure. A public function or procedure can be used in any project. For the detailed discussion of the visibility of project components, see **Section 2.1**.

Class functions and procedures differ from project functions and procedures in their scope, see **Section 2.17**. Project functions and procedures are visible in the entire project. Moreover, public project functions and procedures are visible in all other projects that import their containing project. Class functions and procedures are only visible within their containing class.

2.14.2.1 Internal Functions and Procedures

An internal function or procedure is a subroutine implemented in the Rand Model Designer programming language.

The definition of an internal function or procedure has the following syntax:

```
<internal_function> ::=
  function <function_name>[<formal_parameters>]
    return <return_value_type> is
    <local_variables>
  begin
    <operator_sequence>
  end <function_name>;

<internal_procedure> ::=
  procedure <procedure_name>[<formal_parameters>] is
    <local_variables>
  begin
    <operator_sequence>
  end <procedure_name>;

<function_name> ::= <identifier>

<procedure_name> ::= <identifier>
```

```
<formal_parameters> ::=
  (<parameter_series>{;<parameter_series>})

<parameter_series> ::=
  <parameter_category> <parameter_name>
  {,<parameter_name>}:<parameter_type>

<parameter_category> ::= [in]!out!in out

<parameter_name> ::= <identifier>

<parameter_type> ::= <identifier>

<return_value_type> ::= <identifier>

<local_variables> ::= {<variable_series>}

<variable_series> ::=
  <variable_name >{,<variable_name>}
  :<variable_type>[:=<expression>];

<variable_name> ::= <identifier>

<variable_type> ::= <identifier>

<operator_sequence> ::= {<operator>}
```

Where:

`<function_name>` is the function name unique across all functions and procedures of the containing class or project.

`<procedure_name>` is the procedure name unique across all functions and procedures of the containing class or project.

`<formal_parameters>` is the definition of the formal parameters of the function or procedure.

`<parameter_series>` is a series of formal parameters. All parameters within a series have the same category and type.

`<parameter_category>` is a parameter category with the following possible values: `in`, `out`, and `in out`. `in`, the default, indicates that the parameters of the corresponding series are input parameters. `out` indicates that the parameters of the corresponding series are output parameters. `in out` indicates that the parameters of the corresponding series are input-output parameters.

`<parameter_name>` is a parameter name unique across all parameters of the function or procedure.

`<parameter_type>` is the type of a series of parameters. This can be any available type (see **Section 2.13**).

`<return_value_type>` is the type of the return value of the function. This can be any available type (see **Section 2.13**).

`<local_variables>` is the definition of the local variables of the function or procedure.

`<variable_series>` is a series of local variables. All variables within a series have the same type.

`<variable_name>` is a local variable name unique across all local variables of the function or procedure.

<variable_type> is the type of all variables in a series of local variables. This can be any available type (see **Section 2.13**).

<expression> is an expression (see **Section 2.15**) specifying the initial value of a local variable.

<operator_sequence> is the sequence of operators that implements the function or procedure.

<operator> is a single operator of the Rand Model Designer programming language (see **Section 2.16**).

<identifier> is an identifier (see **Section 6.2**).

Examples:

```
function xlim(x,a: double) return double is
  z: double := abs(x);
begin
  if z > a then
    return a * sign(z);
  else
    return x;
  end if;
end xlim;
```

This function limits the absolute value of the input parameter x by the value of the input parameter a. The local variable z is introduced primarily for demonstration purposes.

```
procedure setup is
begin
  for i in 1..10 loop
    a[i] := i + 2;
  end loop;
end setup;
```

This procedure fills the vector a, which is supposed to be visible at this point.

2.14.2.2 External Functions and Procedures

An external function or procedure is a subroutine implemented outside Rand Model Designer with the use of third party tools such as Visual C++, Delphi, Fortran, etc.

The definition of an external function or procedure within a class or project provides only the specification of the function or procedure. The corresponding implementation must reside in a DLL (dynamic link library) loadable by the executable model.

The definition of an external function or procedure within a class or project has the following syntax:

```
<external_function> ::=
  function <function_name>[<formal_parameters>]
    return <return_value_type> is
    external <path_to_dll> [name <dll_function_name>];

<external_procedure> ::=
  procedure <procedure_name>[<formal_parameters>]
    external <path_to_dll> [name <dll_function_name>];

<function_name> ::= <identifier>

<procedure_name> ::= <identifier>
```

```
<formal_parameters> ::=
  (<parameter_series>{;<parameter_series>})

<parameter_series> ::=
  <parameter_category> <parameter_name>
  {,<parameter_name>}:<parameter_type>

<parameter_category> ::= [in]!out!in out

<parameter_name> ::= <identifier>

<parameter_type> ::= <identifier>

<return_value_type> ::= <identifier>
```

Where:

`<function_name>` is the function name unique across all functions and procedures of the containing class or project.

`<procedure_name>` is the procedure name unique across all functions and procedures of the containing class or project.

`<formal_parameters>` is the definition of the formal parameters of the function or procedure.

`<parameter_series>` is a series of formal parameters. All parameters within a series have the same category and type.

`<parameter_category>` is a parameter category with the following possible values: `in`, `out`, and `in out`. `in`, the default, indicates that the parameters of the corresponding series are input parameters. `out` indicates that the parameters of the corresponding series are output parameters. `in out` indicates that the parameters of the corresponding series are input-output parameters.

`<parameter_name>` is a parameter name unique across all parameters of the function or procedure.

`<parameter_type>` is the type of a series of parameters. This can be any available type (see **Section 2.13**).

`<return_value_type>` is the type of the return value of the function. This can be any available type (see **Section 2.13**).

`<identifier>` is an identifier (see **Section 6.2**).

`<path_to_dll>` is the path to the DLL containing the implementation of the function or procedure. If the path is relative, it is assumed to be relative to the `TMP` folder of the executable model.

`<dll_function_name>` is the name of the DLL function implementing the function or procedure. If omitted, it is assumed to be equal to the name of the external function (`<function_name>`) or procedure (`<procedure_name>`).

Examples:

```
function xlim(x,a: double) return double is
  external "flib1.dll";

procedure inc(out x: double) is
  external "flib1.dll";
```

The DLL function implementing an external function or procedure must be exported from the DLL and must use the `stdcall` calling convention. The parameters of the DLL function must correspond to the parameters specified in the definition of the

external function or procedure. If the external function or procedure has `out` and `in out` parameters, the corresponding parameters of the DLL function must be of type `variant` (variant array).

For example, the external function `xlim` and procedure `inc` defined above can be implemented in Delphi as follows:

```
library flib;
  ...
function xlim(x,a: double): double; stdcall; export;
begin
  if x>a then Result:=a
  else if x<-a then Result:=-a
  else Result:=x;
end {xlim};

procedure inc(var x: double);
begin
  x:=x+1;
end {inc};
  ...
exports xlim,inc;
  ...
end.
```

Vectors, matrices, arrays, and records are passed to DLL functions as `variants`. Vectors and matrices are passed as two-dimensional `[1..N, 1..M]` variant arrays with elements of type double; in case of vectors, `M` equals `1`.

For example, an external function defined as:

```
function addmatrix(in x: matrix; in y: matrix)
  return matrix is external "lib1.dll";
```

can have the following Delphi implementation:

```
library lib1;
function addmatrix(x,y: variant): variant; stdcall;
export;
var
  n,m: integer;
  i,j: integer;
begin
  Result:=Unassigned;
  n:=VarArrayHighBound(x,1);
  m:=VarArrayHighBound(x,2);
  if n<>VarArrayHighBound(y,1) then exit;
  if m<>VarArrayHighBound(y,2) then exit;
  Result:=VarArrayCreate([1,n,1,m],varDouble);
  for i:=1 to n do begin
    for j:=1 to m do begin
      Result[i,j]:=x[i,j]+y[i,j];
    end {for};
  end {for};
end {addmatrix};
exports
addmatrix name 'addmatrix';
begin
end.
```

2.15 EXPRESSIONS

An *expression* is a formula for calculating a new value from one or more existing values. An expression consists of *operands*, *operations*, and parentheses. Operands can be literals (literally expressed fixed values), variables, parameters, constants, function calls, etc. Actions performed on operands are called operations.

2.15.1 Operations

Rand Model Designer supports the following operations:

"+" – addition of integer numbers, floating-point numbers, matrices, and vectors, or concatenation of strings. Examples:

```
1 + 2
5.4 + 6.8
[1;2;3] + [5;8;9]
"abc" + "def"
```

"-" – subtraction of integer numbers, floating-point numbers, matrices, and vectors. Examples:

```
5 - 1
5.4 - 6.8
[1;2;3] - [5;8;9]
```

"*" – multiplication of integer numbers, floating-point numbers, and matrices, or multiplication by a scalar for matrices and vectors. Examples:

```
5 * 2
7.4 * 8.3
A[2,4;5,6] * B[7,8;1,2]
[1;2;3] * 2
```

"/" – division of floating-point numbers. Example:

```
18.3 / 3.03
```

"div" – integer division of integer numbers. Example:

```
107 div 5
```

"mod" – calculation of the remainder of division of one integer number by another. Example:

```
7 mod 5
```

"**" – raising a number to the power of another number, applicable to integer and floating-point numbers. Examples:

```
5.7 **3
2 **4.5
```

"=" – equality relation, applicable to integer numbers, floating-point numbers, matrices, vectors, characters, strings, boolean values, and enumeration values. Examples:

```
2 = 2
[1, 2; 3, 4] = [1, 2; 3, 4]
"q" = "q"
```

"!=" – inequality relation, applicable to integer numbers, floating-point numbers, matrices, vectors, characters, strings, boolean values, and enumeration values. Examples:

```
2 != 2
[1, 2; 3, 4] != [5, 7; 9, 2]
"q" != "q"
```

”>” – “greater than” relation, applicable to integer numbers, floating-point numbers, and enumeration values. Examples:

```
7 > 2
2.4 > 1.8
```

”<” – “less than” relation, applicable to integer numbers, floating-point numbers, and enumeration values. Examples:

```
7 < 2
2.4 < 1.8
```

”>=” – “greater than or equal to” relation, applicable to integer numbers, floating-point numbers, and enumeration values. Examples:

```
7 >= 7
Pi >= 3.1
```

”<= ” – “less than or equal to” relation, applicable to integer numbers, floating-point numbers, and enumeration values. Examples:

```
7 <= 7
Pi <= 3.1
```

”and” – logical “and” for boolean values, bitwise “and” for integer numbers. Examples:

```
false and true
12 and 13
```

”or” – logical “or” for boolean values, bitwise “or” for integer numbers. Examples:

```
false or true
12 or 13
```

”not” – logical “not” for boolean values, bitwise “not” for integer numbers. Examples:

```
not true
not 13
```

Conditional selection operation, which has the following syntax:

```
<conditional_selection> ::=
  if <selector_expression>
  then <expression_if_true> else <expression_if_false>
```

The conditional selection operation works as follows. First, <selector_expression> is evaluated. If the result is true, <expression_if_true> is evaluated. Otherwise <expression_if_false> is evaluated. The result of the operation is the result of whichever expression is evaluated last – <expression_if_true> or <expression_if_false>. Example:

```
if a >= 0 then 1 else -1
```

2.15.2 Evaluation Order

The order of evaluation of any individual operation is determined by the operation.

The order of evaluation of an expression involving more than one operation is determined by the precedence of operations, the order of operations in the expression, and the presence or absence of parentheses.

The precedence of operations decreases in the following order:

conditional selection

”**”

”not”

”*”, ”/”, ”div”, ”mod”, ”and”

”+”, ”-”, ”or”

”=”, ”!=”, ”<”, ”>”, ”<”, ”>”, ”<=”, ”>=”

In the absence of parentheses, operations are evaluated in the order of their precedence, from highest to lowest. Operations of equal precedence are evaluated from left to right. For example, the expression

```
20 + 5 * 2 + 40
```

is evaluated in the following order: the multiplication (`5 * 2`), the first addition (`20 + 10`), the second addition (`30 + 40`).

Parentheses override the normal evaluation order so that every sub-expression within parentheses is evaluated before any other operations are applied to it. For example, the expression

```
20 + 5 * (2 + 40)
```

is evaluated in the following order: the second addition (`2 + 40`), the multiplication (`5 * 42`), the first addition (`20 + 210`).

2.16 OPERATORS

An *operator* is an instruction to the model to take some action. A sequence of one or more operators represents a sequence of instructions that are executed in the order of appearance of the corresponding operators in the sequence. Operators are used to implement internal functions and procedures (see **Section 2.14.2.1**) and specify behavior chart state and transition actions (see **Section 2.4**).

Every operator has its own syntax as described below. Every operator ends with a semicolon.

2.16.1 Assignment Operator

The *assignment* operator (`:=`) assigns the value of its right-hand side expression to the variable referenced by its left-hand side expression.

Syntax:

```
<assignment_operator> ::=
  <lhs_expression> := <rhs_expression>;

<lhs_expression> ::= <expression>

<rhs_expression> ::= <expression>
```

Example:

```
y := 6.8 * (2 + sin(x));
```

2.16.2 Procedure Call Operator

The *procedure call* operator calls a procedure with specified arguments (see **Section 2.14**).

Syntax:

```
<procedure_call_operator> ::=
  <procedure_name>[<argument_list>];

<argument_list> ::=
  (<argument>{,<argument>})

<argument> ::= <expression>
```

The argument expressions are evaluated in the order of their appearance in the argument list, from left to right.

Examples:

```
setup(4.3, true, alpha * 2);

matrixtofile(x, "myfile" + ".txt");
```

2.16.3 The if Operator

The `if` operator performs conditional execution of an operator sequence or a conditional choice between two or more operator sequences, executing only one of them.

Syntax:

```
<if_operator> ::=
  if <conditional_expression> then
     <operator_sequence>
  {elsif <conditional_expression> then
     <operator_sequence>}
  [else
     <operator_sequence>]
  end if;
```

A `<conditional_expression>` is an expression that evaluates to `boolean`.

The `if` operator is executed by first evaluating the conditional expression following the `if` keyword. If the result is `true`, the operator sequence following the first `then` keyword is executed. Otherwise, the conditional expression following the `elsif` keyword, if any, is evaluated. If the result is `true`, the operator sequence following the corresponding `then` keyword is executed. Otherwise, the procedure is repeated for every next pair of the `elsif` and `then` keywords until there is no such pair or the conditional expression evaluates to `true`. If all conditional expressions evaluate to `false`, the operator sequence following the `else` keyword, if any, is executed.

Example:

```
if (x >= x2) and (y <= y1) then
   z := true;
elsif (x < x3) and (x > 0) and (y < y2) then
   z := a or b;
elsif (x > x3) and (y < (a * x + b)) then
   z := a and b;
else
   z := false;
end if;
```

2.16.4 The case Operator

The `case` operator executes one of several operator sequences depending on the value of an expression.

Syntax:

```
<case_operator> ::=
  case <selector_expression> of
  {when <option>{!<option>} =>
    <operator_sequence>}
  [when others =>
    <operator_sequence>]
  end case;
```

The `<selector_expression>` is an expression that evaluates to an integer or enumeration type. Every `<option>` is a literal of the same type as the selector expression.

The `case` operator is executed by first evaluating the selector expression. The result of the selector expression is then compared to each `<option>`. If one of the `<options>`s is equal to the result of the selector expression, the operator sequence following the corresponding => label is executed. Otherwise, the operator sequence following the `when others =>` label, if any, is executed.

Example:

```
case x of
  when 1 ! 2 =>
    y := x + 3;
    z := 4;
  when 5 =>
    z := 0;
  when others =>
    y := 0;
end case;
```

2.16.5 The loop Operator

The `loop` operator repeatedly executes an operator sequence.

Syntax:

```
<loop_operator> ::=
  [<loop_specification>]
  loop
    <operator_sequence>
  end loop;
```

```
<loop_specification> ::=
  for <loop_variable> in <range>
  ! while <conditional_expression>

<loop_variable> ::= <identifier>

<range> ::= <lower_bound>..<upper_bound>
```

A `<conditional_expression>` is an expression that evaluates to `boolean`. `<lower_bound>` and `<upper_bound>` are expressions that evaluate to `integer`.

A loop starting with the `for` keyword, the `for` loop, implicitly introduces a *loop variable*. The name of the loop variables is the identifier specified after the `for` keyword, the type of the loop variable is `integer`, the scope of the loop variable is limited to the operator sequence enclosed in the loop. The range of the loop variable is specified by two expressions, each of which must evaluate to `integer`.

The `for` loop is executed by first evaluating the lower and upper bound expressions. The loop variable is then assigned the value of the lower bound and the enclosed operator sequence is executed. Then, the loop variable is incremented and the enclosed operator sequence is executed again. This is repeated for every variable value up to and including the upper bound.

A loop starting with the `while` keyword, the `while` loop, is executed as follows. First, the conditional expression following the `while` keyword is evaluated. If the result is `true`, the enclosed operator sequence is executed once. Then the conditional expression is evaluated again, and if the result is `true`, the enclosed operator sequence is executed again. This is repeated until the conditional expression evaluates to `false`.

A loop may start with the `loop` keyword, that is, have no `<loop_specification>`. Such a loop executes the enclosed operator sequence repeatedly until the `exit` (see **Section 2.16.6**) or `return` operator (see **Section 2.16.7**) is executed.

Examples:

```
for k in 1..10 loop
   z := z + y;
end loop;

while z > 0 loop
   z := z - x;
end loop;
```

2.16.6 The exit Operator

The `exit` operator transfers control out of an enclosing loop either unconditionally or if a certain condition is satisfied.

Syntax:

```
<exit_operator> ::=
  exit [when <conditional_expression>];
```

A `<conditional_expression>` is an expression that evaluates to `boolean`.

If the optional `when` keyword is absent, the `exit` operator unconditionally transfers control out of the current innermost `loop` operator. If the `when` keyword is present, the `exit` operator evaluates the conditional expression following the `when` keyword and, if the result is `true`, transfers control out of the current innermost `loop` operator.

Example:

```
loop
  z := z - x;
  exit when x <= 0;
end loop;
```

2.16.7 The return Operator

The `return` operator transfers control to the invoker of a function, returning a certain value as the result of the function.

Syntax:

```
<return_operator> ::=
  return <expression>;
```

The return operator evaluates the expression following the `return` keyword and returns control to the caller of the function, passing the result of the expression as the result of the function. The expression must evaluate to the type of the return value of the function (see **Section 2.14.2**).

Example:

```
return 4 + exp(x);
```

2.16.8 The send Operator

The `send` operator creates and sends a signal.

Syntax:

```
<send_operator> ::=
  send!^ [broadcast]
    <variable_name>[<argument_list>];

<variable_name> ::= <identifier>

<argument_list> ::=
  (<argument>{,<argument>})

<argument> ::= <expression>
```

The `send` operator evaluates the argument expressions in the order of their appearance in the argument list, and creates a signal using the resulting values as arguments. The operator then sends the created signal to the specified *source variable*; the source variable must be of a signal type, see **Section 2.13.5**. As the result, the signal arrives at all variables connected to the source variable either directly or through other variables. At that moment, if there are behavior chart transitions awaiting a signal on any of these variables including the source variable, all such transitions are triggered in undefined order. The actions of the triggered transitions can access the arguments of the signal as described in **Section 2.13.5**.

If the optional `broadcast` keyword is present in a `send` operator, the signal is sent as a broadcast signal. A broadcast signal differs from an ordinary signal in that it is automatically transmitted not only between connected variables, but also between disconnected variables if they appear on the same structure diagram and have equal names and types. In particular, if a broadcast signal arrives at an input variable or an internal variable of a compound class, it is automatically relayed to all equally named and typed input variables of all contained local objects. Also, if a broadcast signal arrives at an output variable of a local object, it is automatically relayed to all equally named and typed variables of the containing class and other

local objects. The use of broadcast signals helps reduce the number of links in complex models.

Example:

```
send x(10, true, 34.7);

send broadcast x(10, true, 34.7);
```

2.16.9 Optimization Operator

The *optimization* operator finds the variable values that correspond to the minimum or maximum value of a certain expression.

Syntax:

```
<optimization_operator> ::=
  <direction> <double_expression> for
  <variable_specification>{,<variable_specification>}
  [constraints "{"<constraint>;{<constraint>;}"}"];

<direction> ::= minimize!maximize

<variable_specification> ::=
  <variable_name> [in <range>]

<range> ::= <lower_bound>..<upper_bound>

<constraint> ::=
  <double_expression> <= <double_expression>
  !<double_expression> >= <double_expression>
  !<double_expression> = <double_expression>
```

All variables listed after the `for` keyword, the optimization variables, must be of type `double`. All expressions used in an optimization operator must evaluate to `double`. Every optimization variable can be assigned an individual range of feasible values. Arbitrary non-linear constraints can be specified as well.

Using the current values of optimization variables as the initial estimate, the optimization operator optimizes the specified expression. When the optimum is found, the operator assigns the found values back to the respective variables. The numerical methods used by the optimization operator can be configured separately.

Examples:

```
maximize x**2 + y**2 for
  x in 0..Infinity, y in 0..Infinity
  constraints {y <= 3 - x; y >= x + 1;};

minimize x**2+y**2 for x in 1..3, y in 1..2;

minimize (x-1)**2+(y-2)**2 for x, y;
```

$$(x-y)^2+(z+y)^2 \underset{\substack{x\in[-1,1]\\ z\in[-\infty,\infty]}}{\rightarrow} \min()$$

2.17 VISIBILITY RULES

The visibility rules determine the *scope* of every project element. Project elements are classes, variables, parameters, functions, procedures, etc. The scope of an element is the part of the project where that element is available for use or, in other words, visible.

The notion of a scope is closely intertwined with the notion of a *block*. Each of the following entities designates a block:

Project. The elements visible in a project are the classes, constants, declared types, functions, and procedures defined in that project or imported into that project from other projects.

Class. The elements visible in a class are the parameters, variables, constants, functions, procedures, local classes, and local objects defined in that class.

Function or procedure. The elements visible in a function or procedure are the formal parameters and local variables of that function or procedure.

The `for` loop. The element visible in a `for` loop is the loop variable.

A block can enclose other blocks. For example, if a project contains a class, it means that the corresponding project block encloses the corresponding class block.

Using the notion of a block, the visibility rules can be defines as follows.

Suppose there is an element `E1` with the name `N` defined in some block `B1`. `E1` is visible in its defining block `B1` and, usually, all enclosed blocks. However, if some enclosed block `B2` defines another element `E2` with the same name `N`, `E2` *hides* `E1` in `B2`, that is, makes `N` refer to `E2` but not `E1` in `B2` and its enclosed blocks.

For example, suppose there is a project that defines a constant `X` of type `string` and a class `C` which in its turn defines a variable `X` of type `double`. Within the entire project, `X` will refer to the constant of the project except for the class `C` where `X` will refer to the variable of the class.

Unlike the majority of programming languages, Rand Model Designer block elements are not ordered. This means that a definition of an element within a block may reference any other elements defined in that block. Thus, for example, the expression specifying the initial value of a variable of a class may use any functions defined in that class.

CHAPTER 3
Integrated Modeling Environment

This chapter describes the Rand Model Designer integrated modeling environment

3.1 MAIN WINDOW

The main window, see **Figure 22**, appears when the user starts the integrated modeling environment.

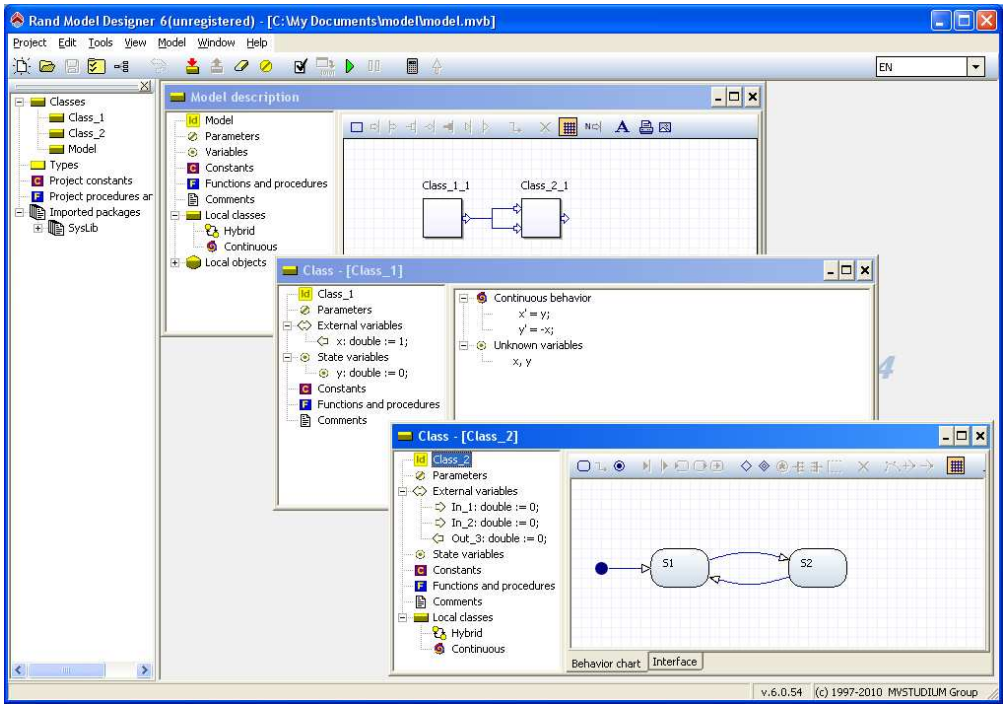


Figure 22 Main window

The main window consists of the main menu, main toolbar, **Project Manager** window, and class editing area. The main menu provides access to most of the functions of the integrated modeling environment. The main toolbar provides shortcuts to some of the most commonly used items of the main menu. The **Project Manager** window displays all components of the currently open project – classes, declared types, constants, functions, procedures, and imported packages – in a tree view. The class editing area serves as a container for class editor windows opened by the user.

3.2 WORKING WITH PROJECTS

The integrated modeling environment allows the user to create, open, save, and rename projects. At any moment, there can be at most one project open. For more on projects, see **Section 2.1**.

❖ **To create a new project:**

1. Click **New Project** on the main toolbar, or
   Choose **Project | New…** from the main menu.
   The **New project** dialog box will appear as shown in **Figure 23**.

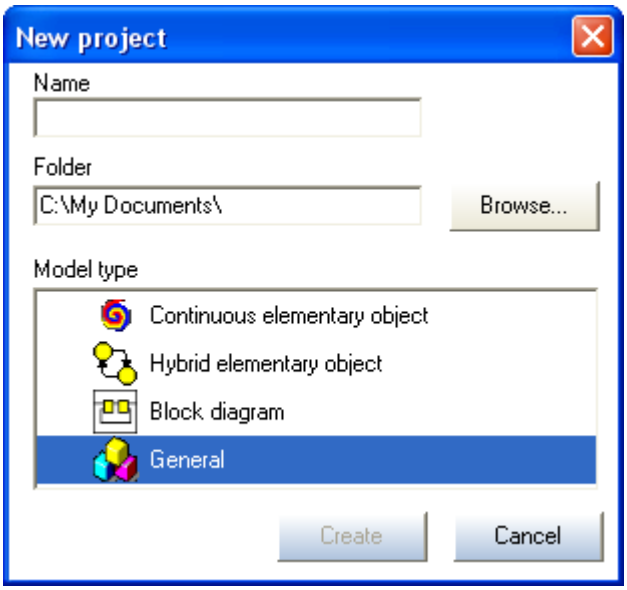


Figure 23 The "New project" dialog box

2. Type the name of the project in the **Name** field.
3. Click **Browse** and browse to the folder that will store the project.
4. Select the model type in the **Model type** list.
   The model type determines the initial stereotype of the automatically created `Model` class (see **Section 2.1** and **Section 2.3**). The stereotype of the `Model` class can be changed later.
5. Click **Create**.

❖ **To open an existing project:**

1. Click **Open project** on the main toolbar, or
   Choose **Project | Open…** from the main menu.
   The **Open** dialog box will appear as shown in **Figure 24**.

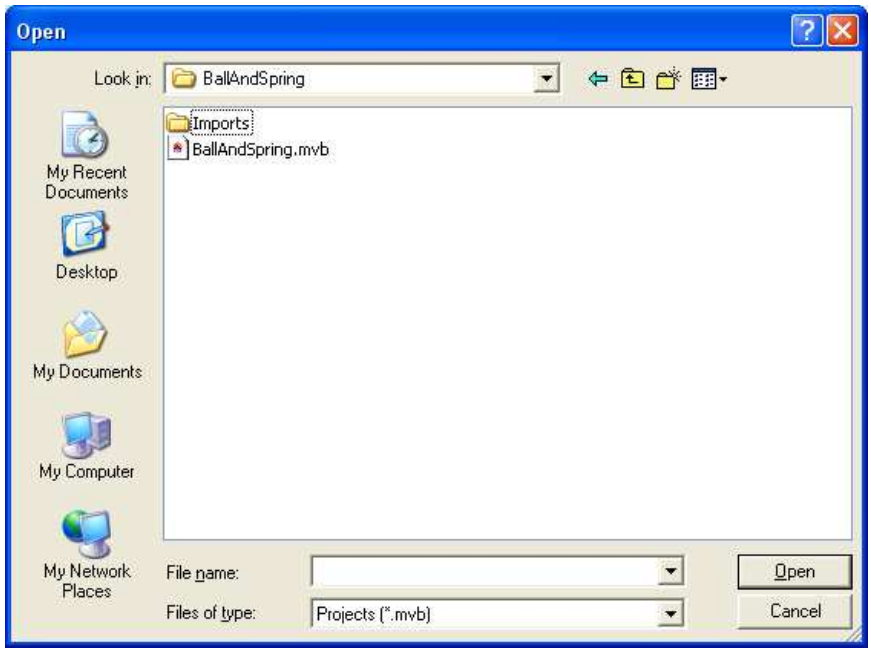


Figure 24 The "Open" dialog box

2. Select the project file you want to open.
3. Click **Open**.

Recently used projects can be opened via the **Project | Reopen** submenu of the main menu.

❖ **To save the currently open project:**

1. Click **Save project** on the main toolbar, or
   Choose **Project | Save** from the main menu.

❖ **To save the currently open project with a new name or location:**

1. Choose **Project | Save as…** from the main menu.
   The **Save project as** dialog box will appear as shown in **Figure 25**.

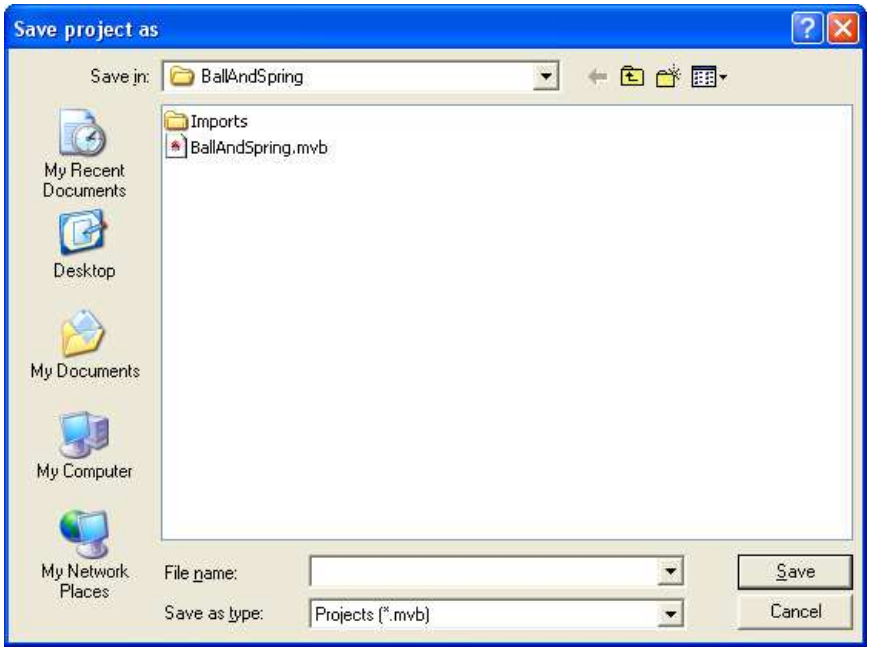


Figure 25 The "Save project as" dialog box

2. Browse to the folder where you want to save the project.
3. Type the new project name in the **File name** field.
4. Click **Save**.

❖ **To rename the currently open project:**

1. Choose **Project | Rename…** from the main menu.
   The **Enter the new project name** dialog box will appear as shown in **Figure 26**.

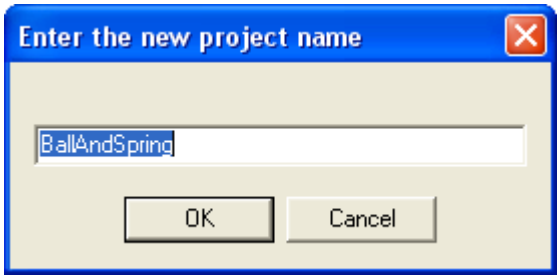


Figure 26 The "Enter the new project name" dialog box

2. Type the new project name.

3. Click **OK**.

❖ **To verify the correctness of the currently open project:**

1. Click **Validate project** on the main toolbar, or
   Choose **Project | Check** from the main menu.
2. If the project has no problems, a success message will be displayed. Otherwise, an error message describing the encountered problems will appear.
3. Click **OK**.

❖ **To close the currently open project:**

1. Choose **Project | Close** from the main menu.

3.3 PROJECT MANAGER

The **Project Manager** window displays a tree view of all components of the currently open project. Project components are classes, declared types, constants, functions, procedures, and imported packages, see **Section 2.1**. The **Project Manager** window looks as shown in **Figure 27**.

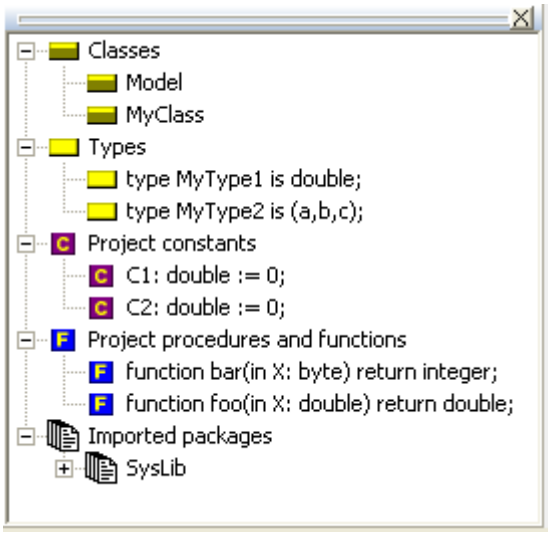


Figure 27 The "Project Manager" window

The **Project Manager** window only exist when there is a project open in the integrated modeling environment. If present, the **Project Manager** window can be resized or toggled on and off.

❖ **To resize the "Project Manager" window:**

1. Click and drag the right edge of the **Project Manager** window.

❖ **To toggle the "Project Manager" window:**

1. Click **Show Project Manager** on the main toolbar, or
   Choose **View | Project Manager** from the main menu.

The tree displayed in the **Project Manager** window contains five top-level nodes: **Classes**, **Types**, **Project constants**, **Project procedures and functions**, and **Imported packages**, each representing a group of project components. Within each group, components are sorted in alphabetical order.

Using the **Project Manager** window, the user creates, modifies, and deletes project components, imports other projects into the currently open project, and exports components of the currently open project into other projects.

The following subsections describe the **Project Manager** window functions specific to each group of project components as well as the functions common to all components.

3.3.1 Classes

For detailed information on classes, see **Section 2.2**.

❖ **To create a new class:**

1. Right-click the **Classes** node in the **Project Manager** window and choose **New class...** from the popup menu.
   The **New Class** dialog box will appear as shown in **Figure 28**.

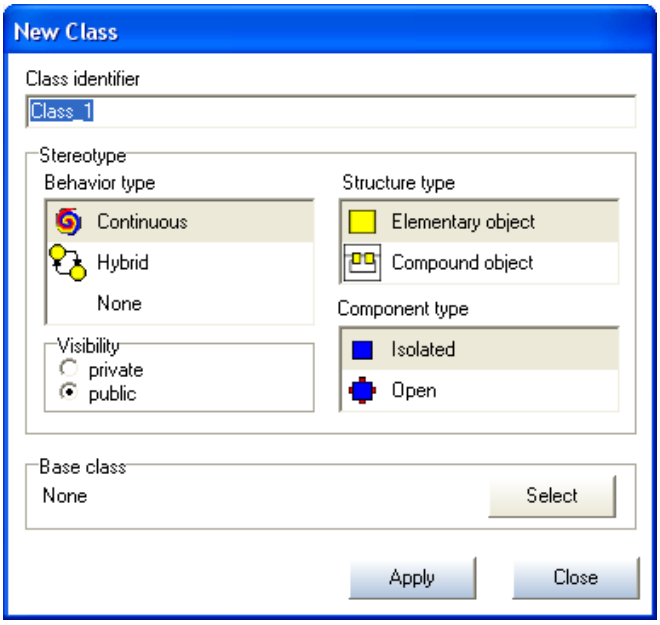


Figure 28 The "New Class" dialog box

2. Type the name of the new class in the **Class identifier** field.
3. Specify the stereotype of the new class by selecting appropriate items in the **Behavior type**, **Structure type**, **Component type**, and **Visibility** lists. You will be able change the stereotype later. For more on the class stereotype, see **Section 2.3**.
4. If it is necessary to specify the base class of the new class, click **Select**.
   The **Choose class** dialog box will appear as shown in **Figure 29**.
   Select the required base class in the **Choose class** dialog box and click **OK**.

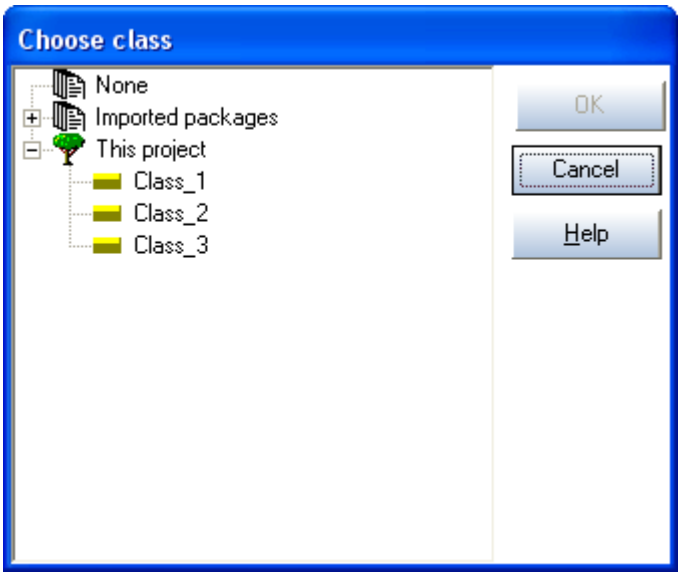


Figure 29 The "Choose class" dialog box

5. In the **New Class** dialog box, click **Apply**.

❖ **To open the class editor window for a class:**

1. Double-click the class in the **Project Manager** window, or
   Right-click the class in the **Project Manager** window and choose **Edit…** from the popup menu.
   The class editor window will appear (see **Section 3.4**).

❖ **To make a class abstract or concrete:**

1. Right-click the class in the **Project Manager** window and choose **Abstract class** from the popup menu.
   This will toggle the flag indicating whether the class is abstract.

❖ **To change the visibility of a class:**

1. Right-click the class in the **Project Manager** window and choose **Public** or **Private** from the popup menu.
   This will set the visibility of the class to the selected value.

3.3.2 Declared Types

For detailed information on declared types, see **Section 2.13.6**.

❖ **To create a new declared type:**

1. Right-click the **Types** node in the **Project Manager** window and choose **New type…** from the popup menu.
   The **User type editing** dialog box will appear as shown in **Figure 30**.

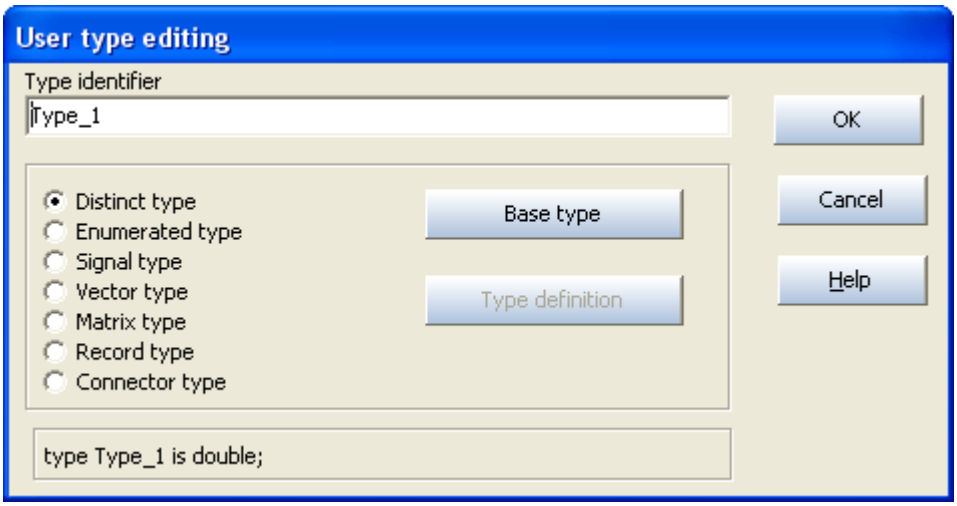


Figure 30 The "User type editing" dialog box

2. Type the name of the new declared type in the **Type identifier** field.
3. If you want the new declared type to act as a synonym for the `double`, `integer`, `short`, `byte`, `boolean`, `string`, or `char` type or another declared type, make sure the **Distinct type** option is checked and click **Base type**.
   Otherwise, select the option that corresponds to the category of the type for which the new declared type will act as a synonym.
   The **Choose type** dialog box will appear.
4. Use the **Choose type** dialog box (see **Section 3.5**) to specify the type for which the new declared type will act as a synonym. When done, click **OK** in the **Choose type** dialog box.
5. Click **OK** in the **User type editing** dialog box.

❖ **To modify an existing declared type:**

1. Double-click the declared type in the **Project Manager** window, or
   Right-click the declared type in the **Project Manager** window and choose **Edit...** from the popup menu.
   The **User type editing** dialog box will appear as shown in **Figure 30**.
2. If you want to modify the name of the type, type the new name in the **Type identifier** field.
3. If you want to modify the specification of the type, proceed as follows.
   If you want the type to act as a synonym for the `double`, `integer`, `short`, `byte`, `boolean`, `string`, or `char` type or another declared type, make sure the **Distinct type** option is checked and click **Base type**.
   Otherwise, select the option that corresponds to the category of the type for which the new type will act as a synonym.
   The **Choose type** dialog box will appear.
   Use the **Choose type** dialog box (see **Section 3.5**) to specify the type for which the new declared type will act as a synonym. When done, click **OK** in the **Choose type** dialog box.
4. Click **OK** in the **User type editing** dialog box.

3.3.3 Constants

For detailed information on constants, see **Section 2.10**.

❖ **To create a new constant:**

1. Right-click the **Project constants** node in the **Project Manager** window and choose **New constant...** from the popup menu.
   The **Constant** window will appear as shown in **Figure 31**.

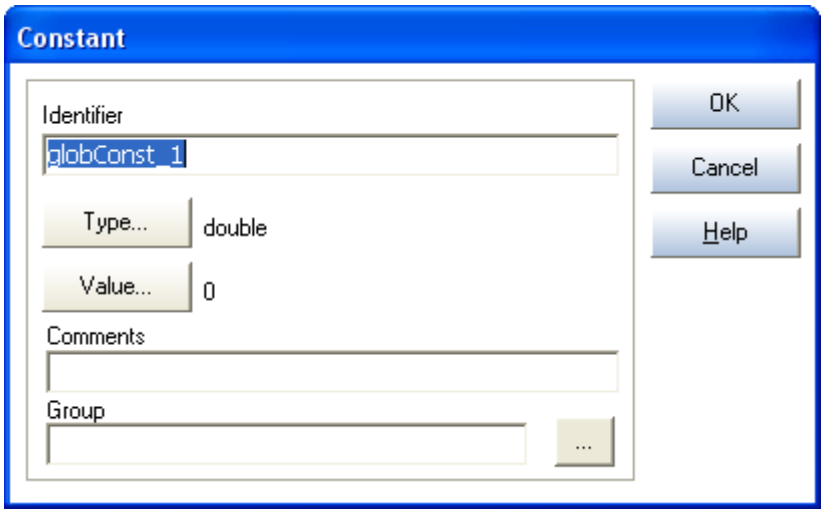


Figure 31 The "Constant" window

2. Type the name of the new constant in the **Identifier** field.
3. To specify the type of the new constant, click **Type...**.
   The **Choose type** dialog box will appear.
   Use the **Choose type** dialog box (see **Section 3.5**) to choose the type of the constant. When done, click **OK** in the **Choose type** dialog box.
4. To specify the value of the new constant, click **Value...**.
   If the type of the constant is an enumeration, the **Choose enumerated value** dialog box will appear as shown in **Figure 32**. Otherwise, if the type of the constant is a record, the **Component Editor** window will appear as shown in **Figure 33**. Otherwise, the **Text Editor** window will appear as shown in **Figure 34**.

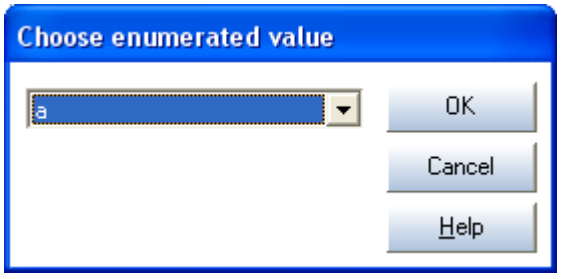


Figure 32 The "Choose enumerated value" dialog box

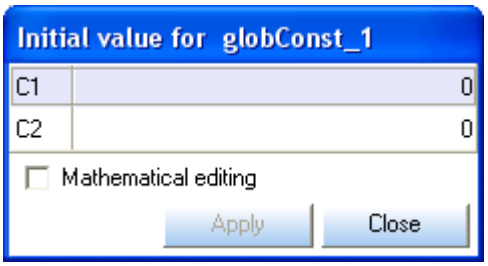


Figure 33 Editing the value of a constant in the "Component Editor" window

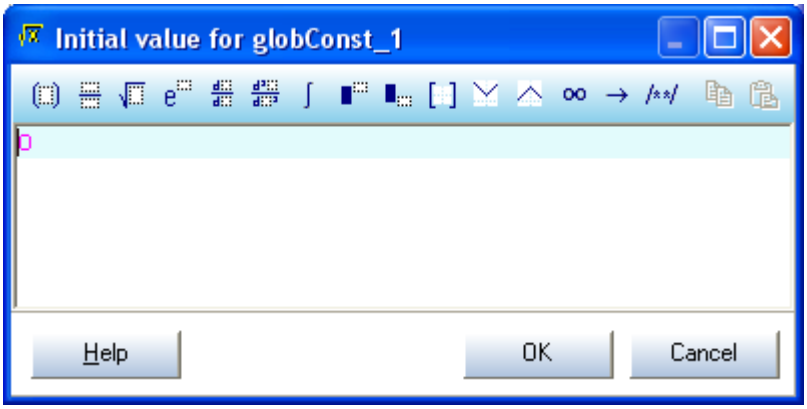


Figure 34 Editing the value of a constant in the "Text Editor" window

5. If the type of the constant is an enumeration, choose the value of the constant in the **Choose enumerated value** dialog box. When done, click **OK** in the **Choose enumerated value** dialog box.
6. If the type of the constant is a record, use the **Component Editor** window (see **Section 3.8**) to specify the value of the constant. When done, click **Apply** in the **Component Editor** window.
7. If the type of the constant is any other type, use the **Text Editor** window (see **Section 3.7**) to specify the expression for the value of the constant. When done, click **OK** in the **Text Editor** window.
8. If necessary, type the comments for the constant in the **Comments** field.
9. If you want to assign the new constant to one or more groups, click .
   The **Groups** dialog box will appear.
   Use the **Groups** dialog box (see **Section 3.6**) to assign the constant to one or more groups. When done, click **Apply** in the **Groups** dialog box.
10. Click **OK** in the **Constant** dialog box.

❖ **To modify a constant using the "Constant" dialog box:**

1. Double-click the constant in the **Project Manager** window, or
   Right-click the constant in the **Project Manager** window and choose **Edit…** from the popup menu.
   The **Constant** dialog box will appear as shown in **Figure 31**.
2. Use the **Constant** dialog box to modify the constant. When done, click **OK**.

❖ **To modify a constant in-place in the "Project Manager" window:**

1. Click the constant in the **Project Manager** window. After a short while, click the constant again.
   An in-place editing field will appear as shown in **Figure 35**.

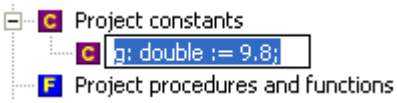


Figure 35 Editing a constant in-place

2. Modify the constant definition in-place.
3. Hit *Enter* or click anywhere in the **Project Manager** window.

3.3.4 User-Defined Functions and Procedures

For detailed information on user-defined functions and procedures, see **Section 2.14.2**.

❖ **To create a new user-defined function or procedure:**

1. Right-click the **Project procedures and functions** node in the **Project Manager** window and choose **New function or procedure…** from the popup menu.
   The **Text Editor** window will appear as shown in **Figure 36**.

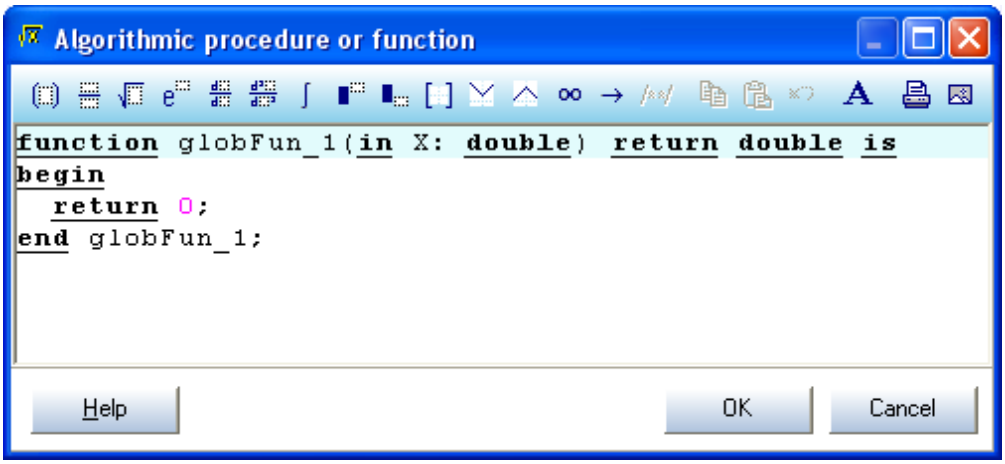


Figure 36 Editing a function or procedure in the "Text Editor" window

2. Use the **Text Editor** window (see **Section 3.7**) to edit the code of the function or procedure.
3. Click **OK**.

❖ **To modify a user-defined function or procedure:**

1. Double-click the function or procedure in the **Project Manager** window, or Right-click the function or procedure in the **Project Manager** window and choose **Edit…** from the popup menu.
   The **Text Editor** window will appear as shown in **Figure 36**.
2. Use the **Text Editor** window (see **Section 3.7**) to edit the code of the function or procedure.
3. Click **OK**.

3.3.5 Deleting Project Components

Any component of the currently open project, except for the `Model` class, can be deleted from the project via the **Project Manager** window.

❖ **To delete a component from the currently open project:**

1. Right-click the component in the **Project Manager** window and choose **Delete** from the popup menu.
   A confirmation message box will appear.
2. Click **Yes**.

3.3.6 Importing and Exporting Project Components

Project components created in one project can be used in other projects. In order for a project component to be usable in other projects, it must be declared as public in its containing project. A project can gain access to the public components of another project by *importing* that other project. An imported project is called an *imported package*. Importing establishes a permanent link between the importing project and the imported package so that all changes made to the imported package affect the importing project.

A project can be imported by any number of other projects. So, if there is a need to share a set of project components between multiple projects, it is recommended to place such components into a dedicated *library* project and have other existing and future projects import that library project; this way the projects importing the library project will always use a single version of each library component.

For more on libraries and import, see **Section 2.1**.

The **Project Manager** window allows the user to import projects into the currently open project. The SysLib library (see **Chapter 5**) is automatically imported into every new project.

❖ **To import a project into the currently open project:**

1. Right-click the **Imported packages** node in the **Project Manager** window and choose **Add imported package…** from the popup menu.
   The **Open** dialog box will appear.
2. Select the project you want to import.
3. Click **Open**.

Sometimes, instead of establishing links between projects, it is more convenient to simply copy some project components from one project to another. The **Project Manager** window provides the export function that allows the user to copy any component from the currently open project into any other project.

❖ **To export a component from the currently open project into another project:**

1. Right-click the component in the **Project Manager** window and choose **Export…** from the popup menu.
   The **Open** dialog box will appear.
2. Select the project where you want to copy the component to.
3. Click **Open**.
   A confirmation message box will be displayed.
4. Click **Yes**.
5. If the export succeeds, a success message will appear.
   Otherwise, an error message describing the encountered problem will appear.

6. Click **OK**.

3.4 CLASS EDITOR

The class editor window allows the user to edit the contents of a class, including the name, parameters, variables, constants, functions, procedures, local classes, behavior chart, system of equations, and structure diagram (see **Section 2.2**).

❖ **To open the class editor for a class:**

1. Double-click the class in the **Project Manager** window, or
   Right-click the class in the **Project Manager** window and choose **Edit...** from the popup menu.

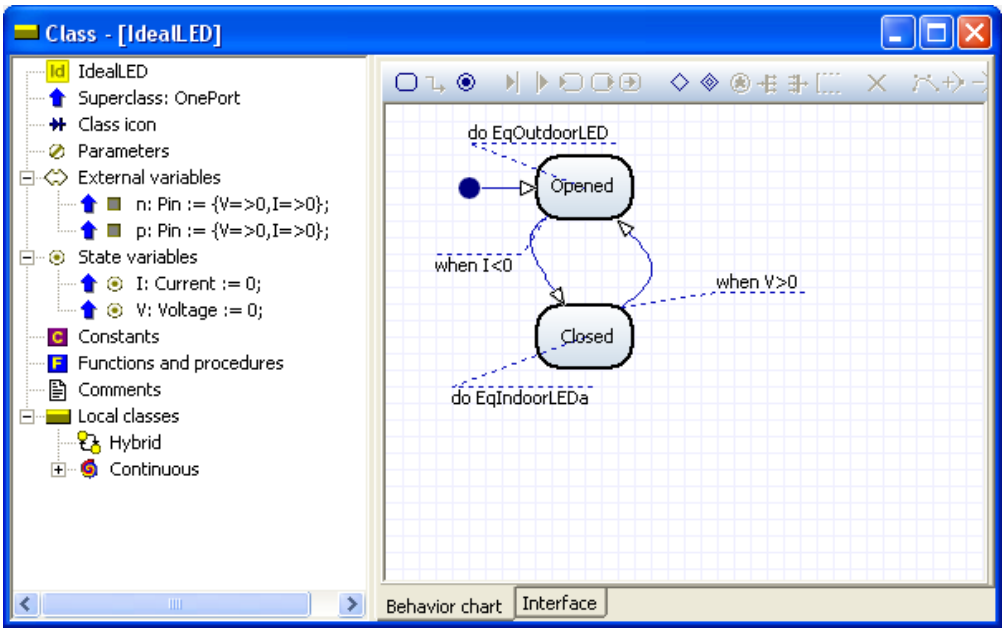


Figure 37 Class editor window

The class editor window looks as shown in **Figure 37**. The class editor window consists of two panels separated by a movable vertical splitter. The left panel displays the tree of properties and elements of the class and is called the class editor tree. The right panel contains a combination of the following editors:

The equation system editor, if the class is continuous

The behavior chart editor, if the class is hybrid

The structure diagram editor, if the class is compound and/or open

Depending on the stereotype of the class (see **Section 2.3**), the right panel can contain one or more editors. If the right panel contains more than one editor, each one appears on a separate tab.

3.4.1 Class Editor Tree

The class editor tree consists of nodes representing groups of elements, such as **Parameters** and **Constants**, as well as nodes representing individual elements, such as the class name, individual parameters, and individual constants. Within a group, elements are sorted in alphabetical order. Inherited elements (see **Section 2.12**) are shown with the ⬆ sign.

The following subsections describe all possible types of elements that can appear in the class editor tree and the associated functions of the class editor.

3.4.1.1 Class Name

The first element of the class editor tree is the class name. The class name uniquely identifies the class within the project.

❖ **To modify the class name:**

1. Double-click the class name node in the class editor tree, or
   Right-click the class name node in the class editor tree and choose **Edit…** from the popup menu.
   An in-place editing field will appear as shown in **Figure 38**.

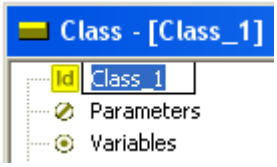


Figure 38 Editing the class name

2. Type the new class name.
3. Hit *Enter* or click anywhere in the class editor window outside the editing field.

3.4.1.2 Class Icon

The class icon is an optional image that, if specified, is shown on every instance of the class.

❖ **To set the class icon:**

1. Right-click the class editor tree and choose **Add icon…** from the popup menu.
   The **Class icon** dialog box will appear as shown in **Figure 39**.

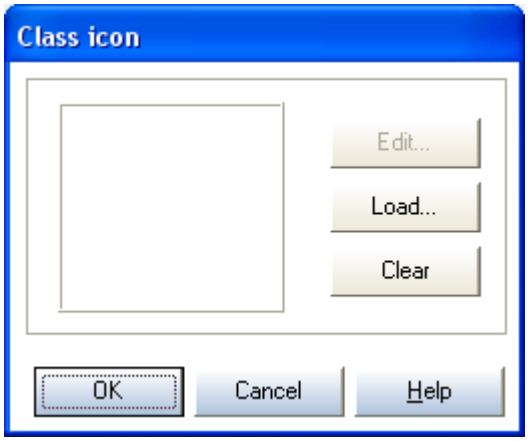


Figure 39 The "Class icon" dialog box (no icon yet)

2. Click **Load….**
   The **Open** dialog box will appear.
3. Select an icon (`.ico`) file.
4. Click **Open**.
   The selected icon will be displayed in the **Class icon** dialog box.
5. Click **OK** in the **Class icon** dialog box.

❖ **To modify the class icon:**

1. Double-click the **Class icon** node in the class editor tree, or
   Right-click the **Class icon** node in the class editor tree and choose **Edit…** from the popup menu.
   The **Class icon** dialog box will appear as shown in **Figure 40**.

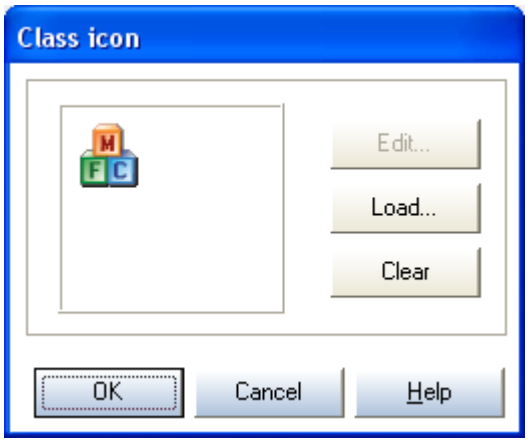


Figure 40 The "Class icon" dialog box (with icon)

2. Click **Load…**.
   The **Open** dialog box will appear.
3. Select an icon (`.ico`) file.
4. Click **Open**.
   The selected icon will be displayed in the **Class icon** dialog box.
5. Click **OK** in the **Class icon** dialog box.

❖ **To clear the class icon:**

1. Double-click the **Class icon** node in the class editor tree, or
   Right-click the **Class icon** node in the class editor tree and choose **Edit…** from the popup menu.
   The **Class icon** dialog box will appear as shown in **Figure 40**.
2. Click **Clear**.
3. Click **OK**.

3.4.1.3 Superclass

The **Superclass** node appears in the class editor tree and displays the name of the base class if the current class has a base class.

The base class is specified upon creation of a class. If a class is created with a base class, the base class can be modified later but cannot be removed. If a class is created without a base class, the base class cannot be added later.

For more on class inheritance, see **Section 2.12**.

❖ **To modify the base class:**

1. Double-click the **Superclass** node in the class editor tree, or
   Right-click the **Superclass** node in the class editor tree and choose **Edit…** from the popup menu.
   The **Choose class** dialog box will appear as shown in **Figure 41**.

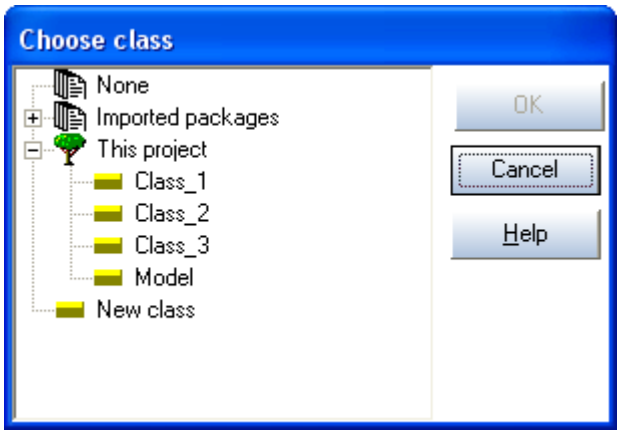


Figure 41 The "Choose class" dialog box

2. Select the new base class and click **OK**.
   A confirmation message box will appear.

3. Click **Yes**.

4. If the recompilation succeeds without problems, the **Superclass** node will start displaying the name of the newly selected base class.
   Otherwise, a window describing the encountered problem will appear, as shown in **Figure 42**.
   In the latter case, you can either try to correct the textual representation of the project (see **Section 3.10**) and recompile the project again by clicking **Retry**, or cancel the operation by clicking **Cancel**.

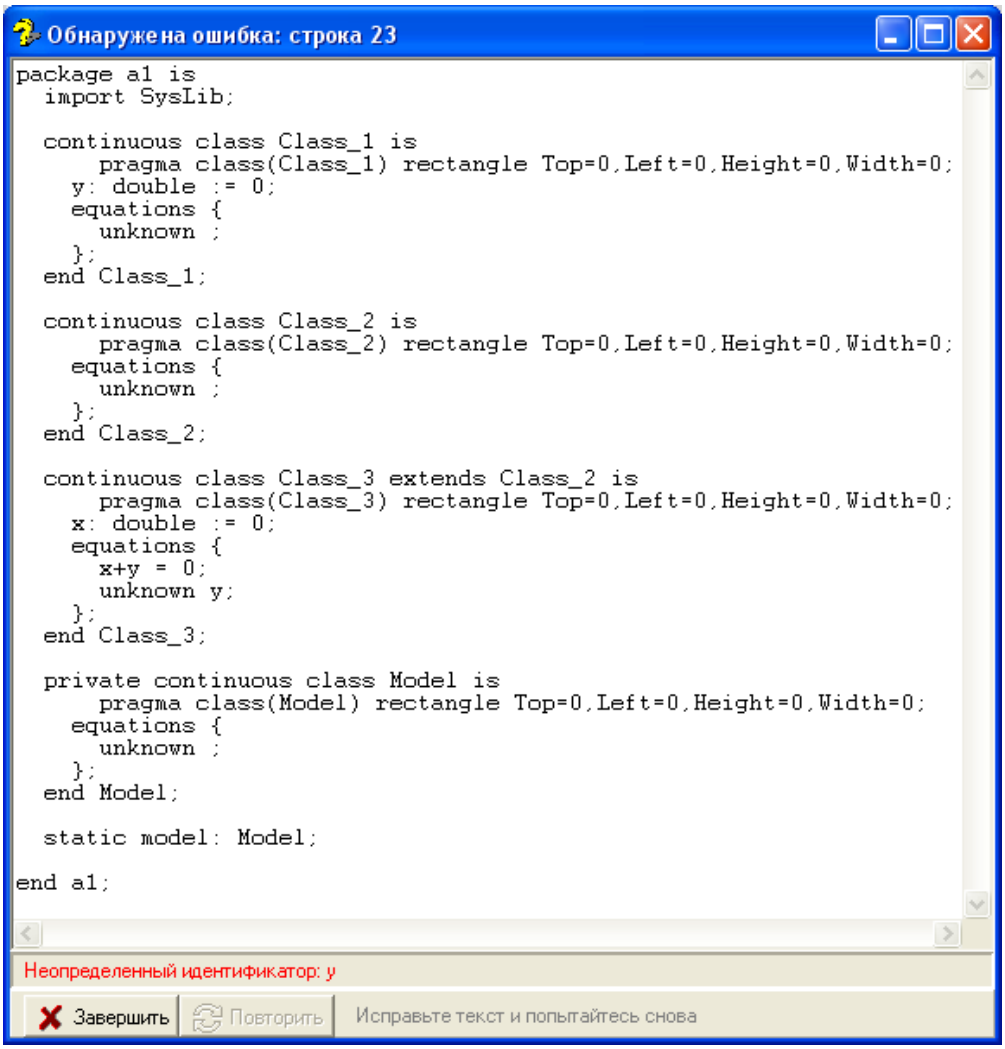


Figure 42 The "Error" window

3.4.1.4 Parameters

The **Parameters** node of the class editor tree represents the parameters of the class. For more on class parameters, see **Section 2.8**.

❖ **To create a new parameter:**

1. Right-click the **Parameters** node in the class editor tree and choose **New parameter...** from the popup menu.
   The **Parameter** dialog box will appear as shown in **Figure 43**.

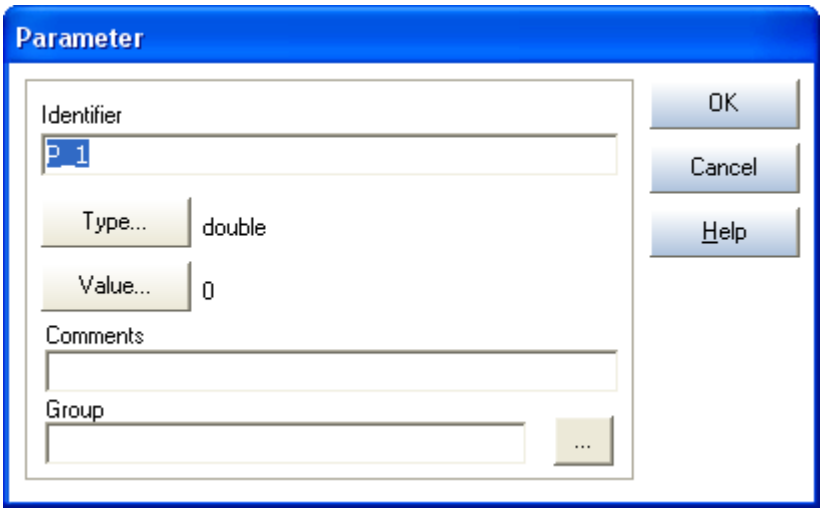


Figure 43 The "Parameter" dialog box

2. Type the name of the new parameter in the **Identifier** field.

3. To specify the type of the new parameter, click **Type…**.
   The **Choose type** dialog box will appear.
   Use the **Choose type** dialog box (see **Section 3.5**) to choose the type of the parameter. When done, click **OK** in the **Choose type** dialog box.

4. To specify the default value of the new parameter, click **Value…**.
   If the type of the parameter is an enumeration, the **Choose enumerated value** dialog box will appear as shown in **Figure 44**. Otherwise, if the type of the parameter is a record, the **Component Editor** window will appear as shown in **Figure 45**. Otherwise, the **Text Editor** window will appear as shown in **Figure 46**.

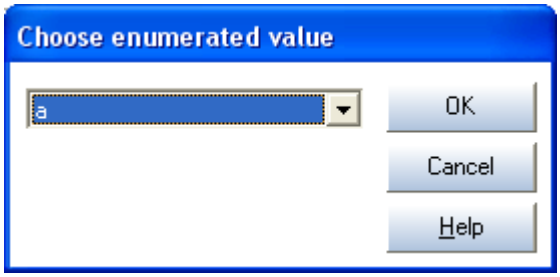


Figure 44 The "Choose enumerated value" dialog box

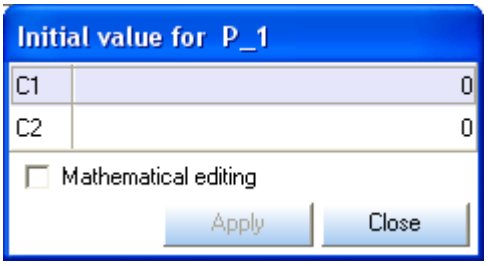


Figure 45 Editing the default value of a parameter in the "Component Editor" window

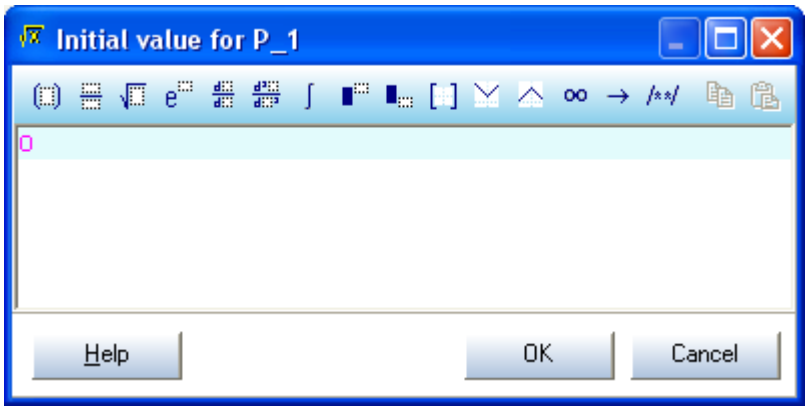


Figure 46 Editing the default value of a parameter in the "Text Editor" window

5. If the type of the parameter is an enumeration, choose the default value of the parameter in the **Choose enumerated value** dialog box. When done, click **OK** in the **Choose enumerated value** dialog box.

6. If the type of the parameter is a record, use the **Component Editor** window (see **Section 3.8**) to specify the default value of the parameter. When done, click **Apply** in the **Component Editor** window.

7. If the type of the parameter is any other type, use the **Text Editor** window (see **Section 3.7**) to specify the expression for the default value of the parameter. When done, click **OK** in the **Text Editor** window.

8. If necessary, type the comments for the parameter in the **Comments** field.

9. If you want to assign the new parameter to one or more groups, click .
   The **Groups** dialog box will appear.
   Use the **Groups** dialog box (see **Section 3.6**) to assign the parameter to one or more groups. When done, click **Apply** in the **Groups** dialog box.

10. Click **OK** in the **Parameter** dialog box.

❖ **To modify a parameter using the "Parameter" dialog box:**

1. Double-click the parameter in the class editor tree, or
   Right-click the parameter in the class editor tree and choose **Edit...** from the popup menu.
   The **Parameter** dialog box will appear as shown in **Figure 43**.

2. Use the **Parameter** dialog box to modify the parameter. When done, click **OK**.

❖ **To modify a parameter in-place in the class editor tree:**

1. Click the parameter in the class editor tree. After a short while, click the parameter again.
   An in-place editing field will appear as shown in **Figure 47**.

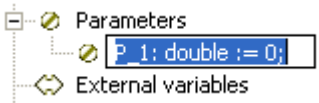


Figure 47 Editing a parameter in-pace

2. Modify the parameter definition in-place.

3. Hit *Enter* or click anywhere in the class editor.

❖ **To convert a parameter into a variable or a constant:**

1. Drag the parameter to the **External variables**, **State variables**, **Variables**, or **Constants** node of the class editor tree.

3.4.1.5 Variables

If a class is open, its class editor tree contains the **External variables** and **State variables** nodes representing the external and internal variables of the class, respectively. If a class is closed, its class editor tree contains the **Variables** node representing the internal variables of the class. For more on class variables, see **Section 2.9**.

❖ **To create a new internal variable:**

1. Right-click the **Internal variables** or **Variables** node in the class editor tree and choose **New internal variable…** or **New variable…** from the popup menu. The **Internal variable** dialog box will appear as shown in **Figure 48**.

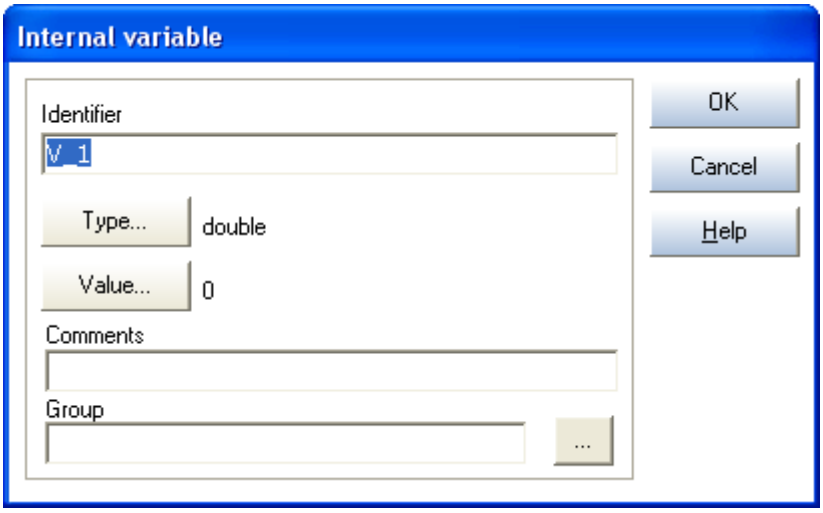


Figure 48 The "Internal variable" dialog box

2. Type the name of the new variable in the **Identifier** field.
3. To specify the type of the new variable, click **Type…**. The **Choose type…** dialog box will appear. Use the **Choose type** dialog box (see **Section 3.5**) to choose the type of the variable. When done, click **OK** in the **Choose type** dialog box.
4. To specify the default initial value of the new variable, click **Value…**. If the type of the variable is an enumeration, the **Choose enumerated value** dialog box will appear as shown in **Figure 49**. Otherwise, if the type of the variable is a record, the **Component Editor** window will appear as shown in **Figure 50**. Otherwise, the **Text Editor** window will appear as shown in **Figure 51**.

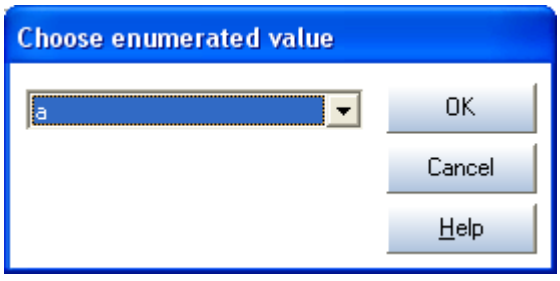


Figure 49 The "Choose enumerated value" dialog box

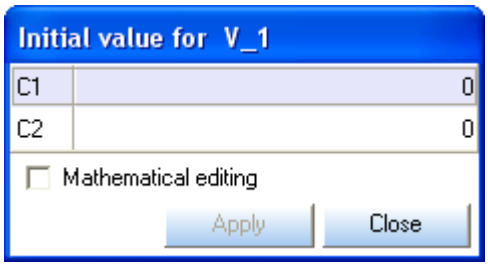


Figure 50 Editing the default initial value of a variable in the "Component Editor" window

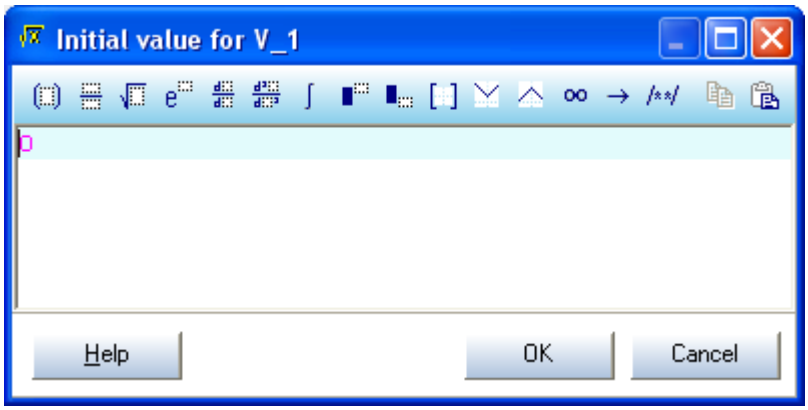


Figure 51 Editing the default initial value of a variable in the "Text Editor" window

5. If the type of the variable is an enumeration, choose the default initial value in the **Choose enumerated value** dialog box. When done, click **OK** in the **Choose enumerated value** dialog box.
6. If the type of the variable is a record, use the **Component Editor** window (see **Section 3.8**) to specify the default initial value of the variable. When done, click **Apply** in the **Component Editor** window.
7. If the type of the variable is any other type, use the **Text Editor** window (see **Section 3.7**) to specify the expression for the default initial value of the variable. When done, click **OK** in the **Text Editor** window.
8. If necessary, type the comments for the variable in the **Comments** field.
9. If you want to assign the new variable to one or more groups, click .
   The **Groups** dialog box will appear.
   Use the **Groups** dialog box (see **Section 3.6**) to assign the variable to one or more groups. When done, click **Apply** in the **Groups** dialog box.
10. Click **OK** in the **Internal variable** window.

❖ **To create a new external variable:**

1. Right-click the **External variables** node in the class editor tree and choose **New external variable…** from the popup menu.
   The **External variable** dialog box will appear as shown in **Figure 52**.

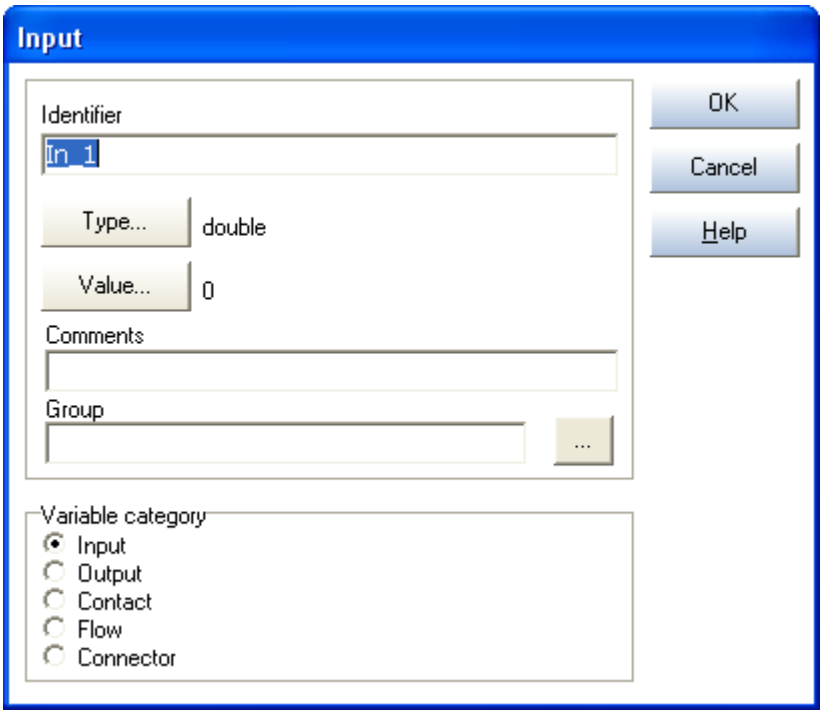


Figure 52 The "External variable" dialog box

2. Choose the category of the new external variable in the **Variable category** list.

3. Type the name of the new variable in the **Identifier** field.

4. To specify the type of the new variable, click **Type…**.
   The **Choose type…** dialog box will appear.
   Use the **Choose type** dialog box (see **Section 3.5**) to choose the type of the variable. When done, click **OK** in the **Choose type** dialog box.

5. To specify the default initial value of the new variable, click **Value…**.
   If the type of the variable is an enumeration, the **Choose enumerated value** dialog box will appear as shown in **Figure 53**. Otherwise, if the type of the variable is a record or connector, the **Component Editor** window will appear as shown in **Figure 54**. Otherwise, the **Text Editor** window will appear as shown in **Figure 55**.

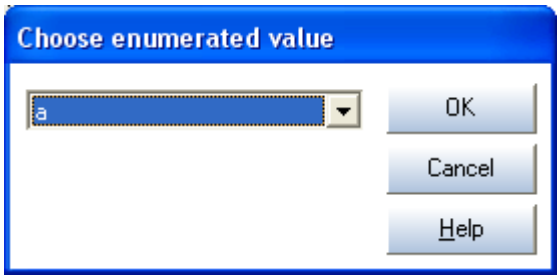


Figure 53 The "Choose enumerated value" dialog box

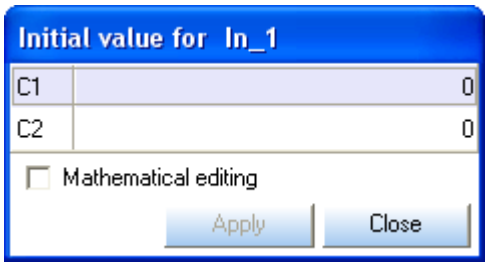


Figure 54 Editing the default initial value of a variable in the "Component Editor" window

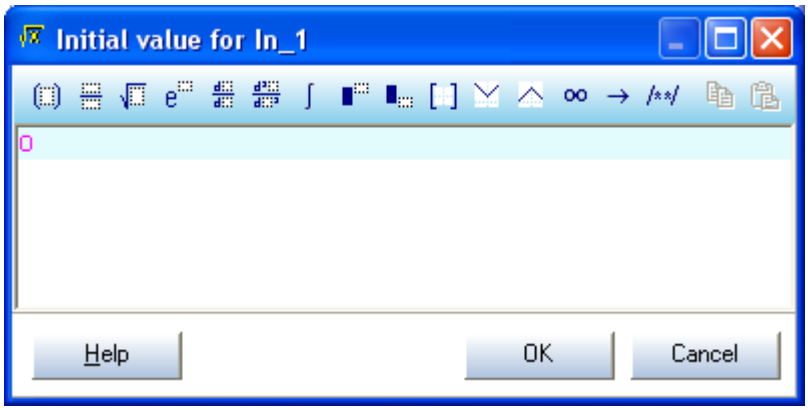


Figure 55 Editing the default initial value of a variable in the "Text Editor" window

6. If the type of the variable is an enumeration, choose the default initial value in the **Choose enumerated value** dialog box. When done, click **OK** in the **Choose enumerated value** dialog box.

7. If the type of the variable is a record or connector, use the **Component Editor** window (see **Section 3.8**) to specify the default initial value of the variable. When done, click **Apply** in the **Component Editor** window.

8. If the type of the variable is any other type, use the **Text Editor** window (see **Section 3.7**) to specify the expression for the default initial value of the variable. When done, click **OK** in the **Text Editor** window.

9. If necessary, type the comments for the variable in the **Comments** field.

10. If you want to assign the new variable to one or more groups, click .
    The **Groups** dialog box will appear.
    Use the **Groups** dialog box (see **Section 3.6**) to assign the variable to one or more groups. When done, click **Apply** in the **Groups** dialog box.

11. Click **OK** in the **External variable** dialog box.

❖ **To modify a variable using the "Internal variable" or "External variable" dialog box:**

1. Double-click the variable in the class editor tree, or
   Right-click the variable in the class editor tree and choose **Edit…** from the popup menu.
   The **Internal variable** or **External variable** dialog box will appear as shown in **Figure 48** or **Figure 52**, respectively.

2. Use the appeared dialog box to modify the variable. When done, click **OK**.

❖ **To modify a variable in-place in the class editor tree:**

1. Click the variable in the class editor tree. After a short while, click the variable again.
   An in-place editing field will appear as shown in **Figure 56**.

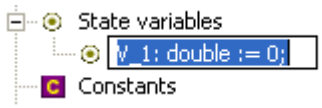


Figure 56 Editing a variable in-place

2. Modify the variable definition in-place.
3. Hit *Enter* or click anywhere in the class editor.

If the initial value of a variable is specified by a static expression, the variable can be converted into a parameter or a constant.

❖ **To convert a variable into a parameter or a constant:**

1. Drag the variable to the **Parameters** node or the **Constants** node of the class editor tree.

3.4.1.6 Constants

The **Constants** node of the class editor tree represents the constants of the class. For more on class constants, see **Section 2.10**.

❖ **To create a new constant:**

1. Right-click the **Constants** node in the class editor tree and choose **New constant...** from the popup menu.
   The **Constant** dialog box will appear as shown in **Figure 57**.

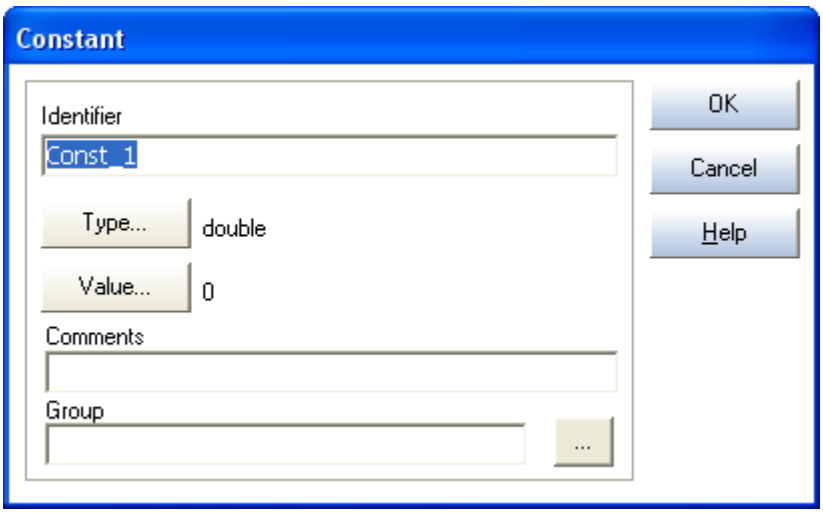


Figure 57 The "Constant" dialog box

2. Type the name of the new constant in the **Identifier** field.
3. To specify the type of the new constant, click **Type...**.
   The **Choose type** dialog box will appear.
   Use the **Choose type** dialog box (see **Section 3.5**) to choose the type of the constant. When done, click **OK** in the **Choose type** dialog box.
4. To specify the value of the new constant, click **Value...**.
   If the type of the constant is an enumeration, the **Choose enumerated value** dialog box will appear as shown in **Figure 58**. Otherwise, if the type of the

constant is a record, the **Component Editor** window will appear as shown in **Figure 59**. Otherwise, the **Text Editor** window will appear as shown in **Figure 60**.

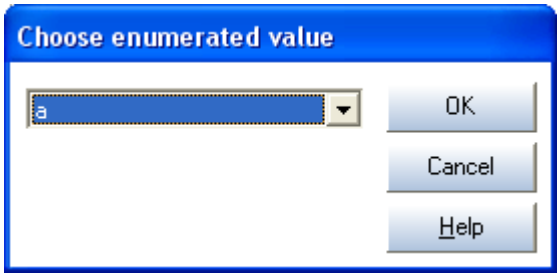


Figure 58 The "Choose enumerated value" dialog box

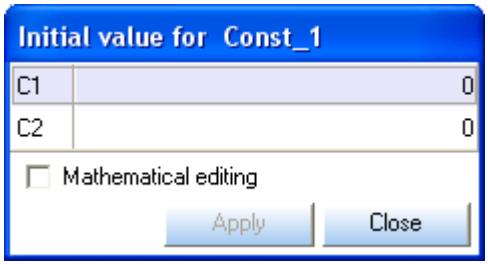


Figure 59 Editing the value of a constant in the "Component Editor" window

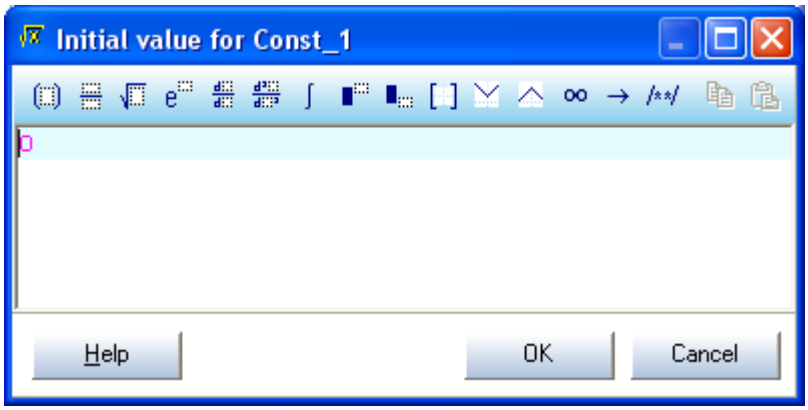


Figure 60 Editing the value of a constant in the "Text Editor" window

5. If the type of the constant is an enumeration, choose the value of the constant in the **Choose enumerated value** dialog box. When done, click **OK** in the **Choose enumerated value** dialog box.

6. If the type of the constant is a record, use the **Component Editor** window (see **Section 3.8**) to specify the value of the constant. When done, click **Apply** in the **Component Editor** window.

7. If the type of the constant is any other type, use the **Text Editor** window (see **Section 3.7**) to specify the expression for the value of the constant. When done, click **OK** in the **Text Editor** window.

8. If necessary, type the comments for the constant in the **Comments** field.

9. If you want to assign the new constant to one or more groups, click .
   The **Groups** dialog box will appear.
   Use the **Groups** dialog box (see **Section 3.6**) to assign the constant to one or more groups. When done, click **Apply** in the **Groups** dialog box.

10. Click **OK** in the **Constant** dialog box.

❖ **To modify a constant using the "Constant" dialog box:**

1. Double-click the constant in the class editor tree, or
   Right-click the constant in the class editor tree and choose **Edit…** from the popup menu.
   The **Constant** dialog box will appear as shown in **Figure 57**.

2. Use the **Constant** dialog box to modify the constant. When done, click **OK**.

❖ **To modify a constant in-place in the class editor tree:**

1. Click the constant in the class editor tree. After a short while, click the constant again.
   An in-place editing field will appear as shown in **Figure 61**.

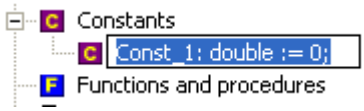


Figure 61 Editing a constant in-place

2. Modify the constant definition in-place.

3. Hit *Enter* or click anywhere in the class editor.

❖ **To convert a constant into a parameter or a variable:**

1. Drag the constant to the **Parameters**, **External variables**, **State variables**, or **Variables** node of the class editor tree.

3.4.1.7 Functions and Procedures

The **Functions and procedures** node of the class editor tree represents the functions and procedures of the class. For more on class functions and procedures, see **Section 2.14.2**.

❖ **To create a new function or procedure:**

1. Right-click the **Functions and procedures** node in the class editor tree and choose **New function or procedure…** from the popup menu.
   The **Text Editor** window will appear as shown in **Figure 62**.

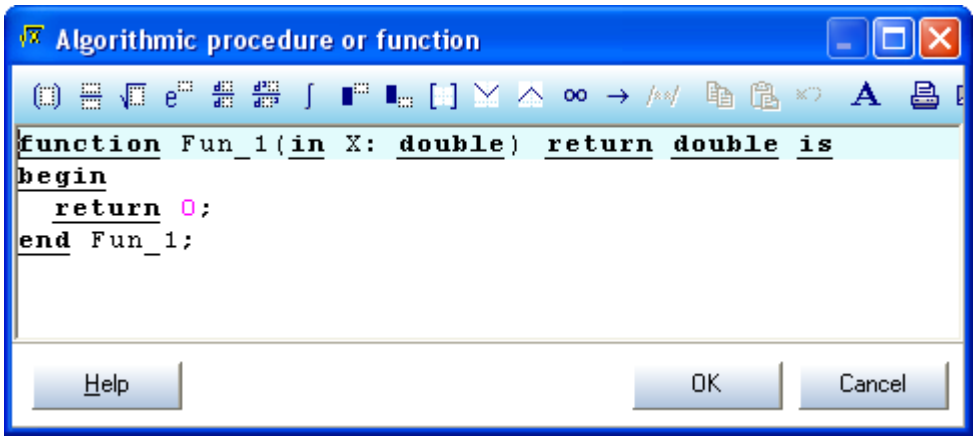


Figure 62 Editing a function or procedure in the "Text Editor" window

2. Use the **Text Editor** window (see **Section 3.7**) to edit the code of the function or procedure.

3. Click **OK**.

❖ **To modify a function or procedure:**

1. Double-click the function or procedure in the class editor tree, or
   Right-click the function or procedure in the class editor tree and choose **Edit…** from the popup menu.
   The **Text Editor** window will appear as shown in **Figure 62**.

2. Use the **Text Editor** window (see **Section 3.7**) to edit the code of the function or procedure.

3. Click **OK**.

3.4.1.8 Local Classes

If a class is hybrid, its class editor tree contains the **Local classes** node that represents the local classes of the class. The **Local classes** node, if present, consists of two sub-nodes, **Hybrid** and **Continuous**, representing the hybrid and continuous local classes, respectively.

For more on local classes, see **Section 2.11**.

❖ **To create a new hybrid local class:**

1. Right-click the **Hybrid** node in the class editor tree and choose **New local hybrid class…** from the popup menu.
   A new local class will be added to the **Hybrid** node.

❖ **To create a new continuous local class:**

1. Right-click the **Continuous** node in the class editor tree and choose **New local continuous class…** from the popup menu.
   A new local class will be added to the **Continuous** node.

❖ **To open the class editor for a local class:**

1. Double-click the local class in the class editor tree, or
   Right-click the local class in the class editor tree and choose **Edit…** from the popup menu.
   The class editor window for of the corresponding local class will appear.

3.4.1.9 Local Objects

If a class is compound, its class editor tree contains the **Local objects** node that represents the local objects of the class.

Local objects are added to and deleted from the class using the structure diagram editor, see **Section 3.4.4**. Local objects cannot be added or deleted via the class editor tree. The class editor tree only displays the local objects as individual sub-nodes under the **Local objects** node. The sub-nodes of the **Local objects** node allow the user to quickly select local objects and modify their actual parameter values and initial variable values.

For more on local objects, see **Section 2.7.1**.

❖ **To select a local object in the structure diagram editor:**

1. Double-click the corresponding node in the class editor tree.

❖ **To modify the actual parameter values and initial variable values of a local object:**

1. Right-click the corresponding node in the class editor tree and choose **Actual values of parameters and variables…** from the popup menu.
   The **Actual values** dialog box will appear as shown in **Figure 63**.

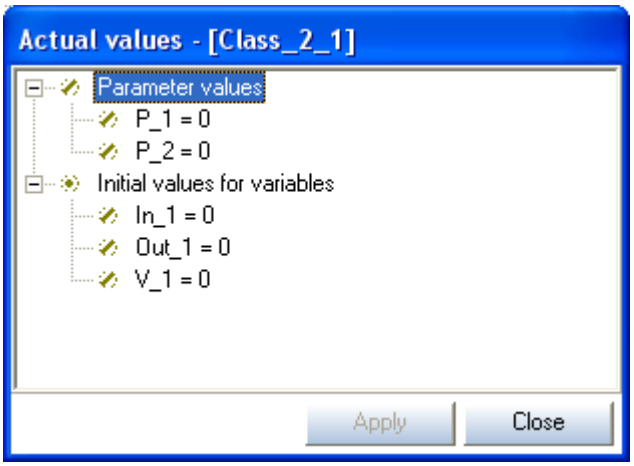


Figure 63 The "Actual values" dialog box

2. To modify the actual value of a parameter or the initial value of a variable, double-click the corresponding item in the **Actual values** dialog box, or right-click the corresponding item in the **Actual values** dialog box and choose **New value…** from the popup menu.
   A value editing dialog box, as shown in **Figure 64**, will appear.

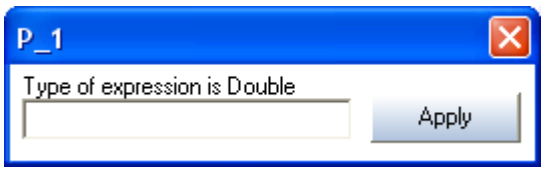


Figure 64 Value editing dialog box

3. Type the expression for the actual parameter value or initial variable value. When done, click **Apply**.
   The **Actual values** dialog box will display the modified parameter or variable with the sign.

4. To restore the actual value of a parameter or the initial value of a variable to the default value, right-click the corresponding item in the **Actual values** dialog box and choose **Restore default value** from the popup menu.

5. Finally, click **Apply** in the **Actual values** dialog box.

3.4.1.10 Comments

The **Comments** node of the class editor tree allows the user to specify the class comments.

❖ **To modify the class comments:**

1. Double-click the **Comments** node in the class editor tree, or
   Right-click the **Comments** node in the class editor tree and choose **Edit…** from the popup menu.
   The **Text Editor** window will appear as shown in **Figure 65**.

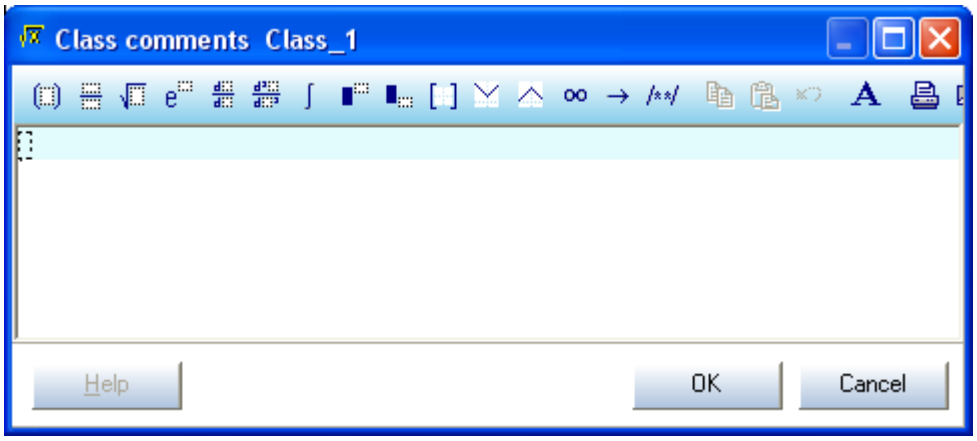


Figure 65 Editing the class comments in the "Text Editor" window

2. Edit the comments in the **Text Editor** window.
3. Click **OK**.

3.4.1.11 Deleting Class Elements

All class elements created by the user can be deleted by the user.

❖ **To delete a class element:**

1. Right-click the element in the class editor tree and choose **Delete** from the popup menu.
   A confirmation message box will be displayed.
2. Click **Yes**.

3.4.1.12 Modifying Class Stereotype

Under certain conditions, the class editor window allows the user to alter the stereotype of the class. The following stereotype conversions are supported:

A continuous class can be converted to an equivalent hybrid class.

A hybrid class whose behavior chart consists of only one continuous activity state can be converted to an equivalent continuous class.

An elementary class can be converted to an equivalent compound class without local objects and links, and vice versa.

An isolated class can be converted to an equivalent open class without external variables, and vice versa.

❖ **To modify the class stereotype:**

1. Right-click anywhere in the class editor tree and choose the desired conversion from the **Transform to** submenu of the popup menu.

3.4.2 Equation System Editor

The equation system editor allows the user to edit the system of equations of a class. For more on the system of equations, see **Section 2.5**.

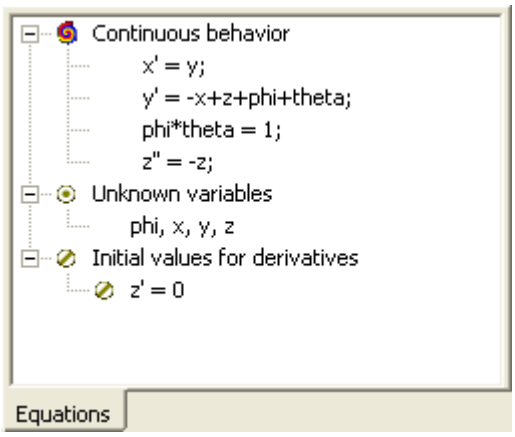


Figure 66 Equation system editor

The equation system editor looks as shown in **Figure 66**. The equations system editor allows the user to:

Edit the equations participating in the system

Edit the declarations of known and unknown variables

Specify the initial values of the first derivatives for those variables whose second derivatives participate in the equations

❖ **To edit the equations participating in the system:**

1. Double-click the **Continuous behavior** node or any of its sub-nodes in the equation system editor, or
   Right-click the **Continuous behavior** node or any of its sub-nodes in the equation system editor and choose **Edit…** from the popup menu.
   The **Text Editor** window will appear as shown in **Figure 67**.

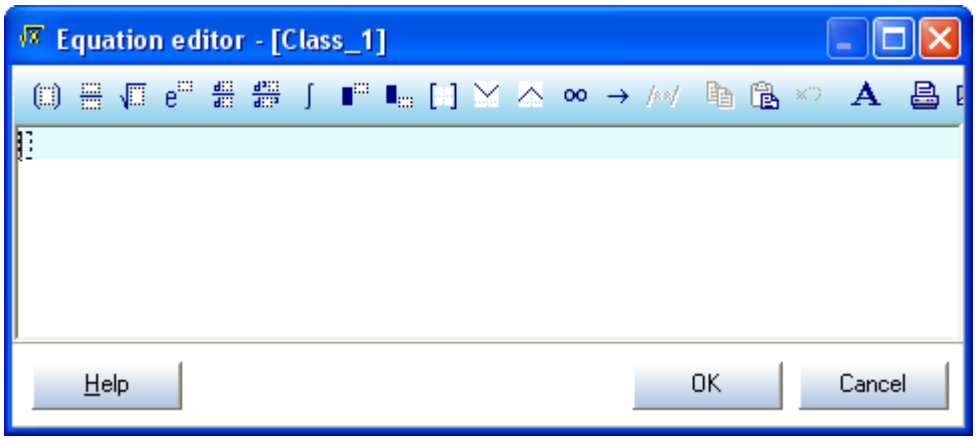


Figure 67 Editing equations in the "Text Editor" window

2. Use the **Text Editor** window (see **Section 3.7**) to edit the equations of the system. The result can look, for example, as shown in **Figure 68**. In the **Text Editor** window, you can specify not only equations but also declarations of known and unknown variables.

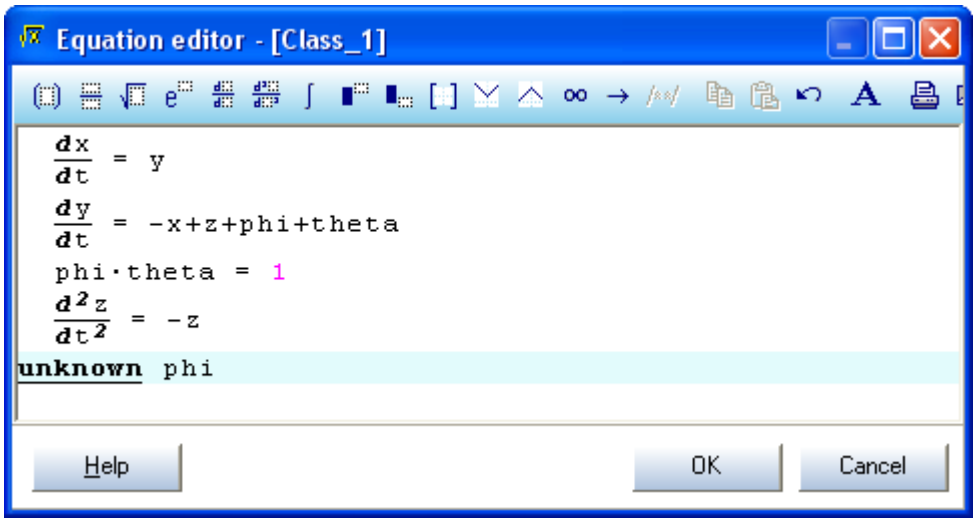


Figure 68 Editing equations, example 1

3. Click **OK** in the **Text Editor** window.

4. If the number of equations is equal to the number of explicitly declared unknown variables, the new system of equations will appear in the equation system editor immediately as shown in **Figure 69**. Otherwise, Rand Model Designer will attempt to select the required number of unknown variables automatically (see **Section 2.5**). If the automatic selection succeeds, the resulting system of equations will appear in the equation system editor immediately as well.

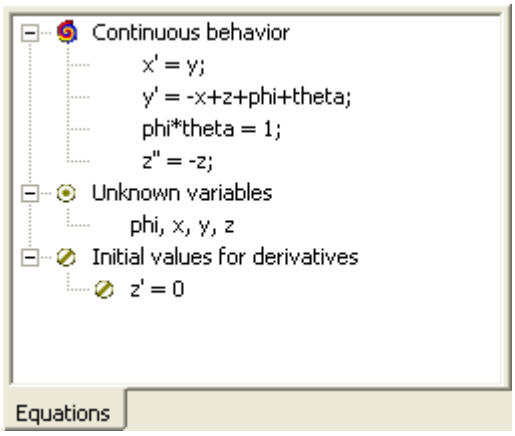


Figure 69 Resulting system of equations, example 1

5. If the automatic selection of unknown variables fails (for example, as in the case of a system of equations shown in **Figure 70**), the **Select unknown variables** dialog box will appear as shown in **Figure 71**.

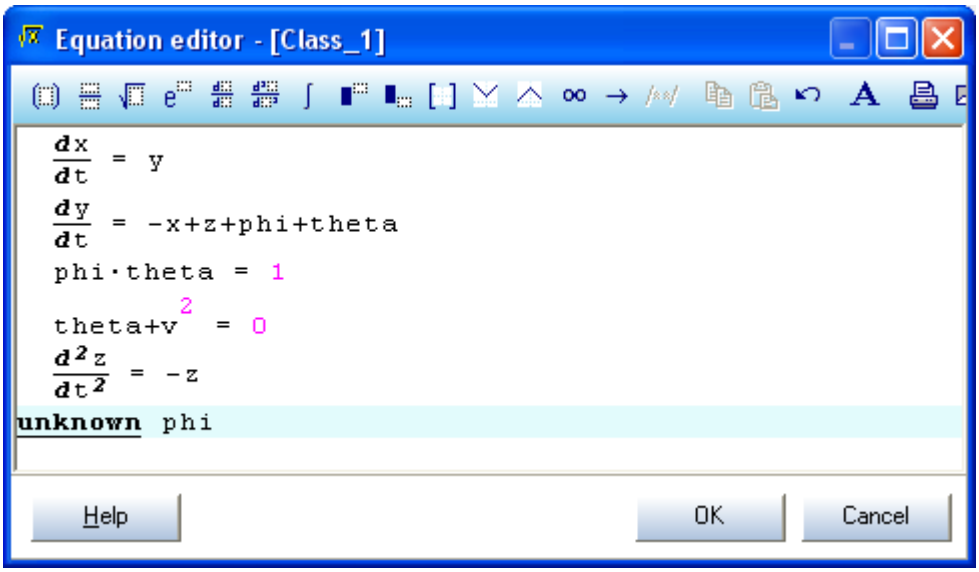


Figure 70 Editing equations, example 2

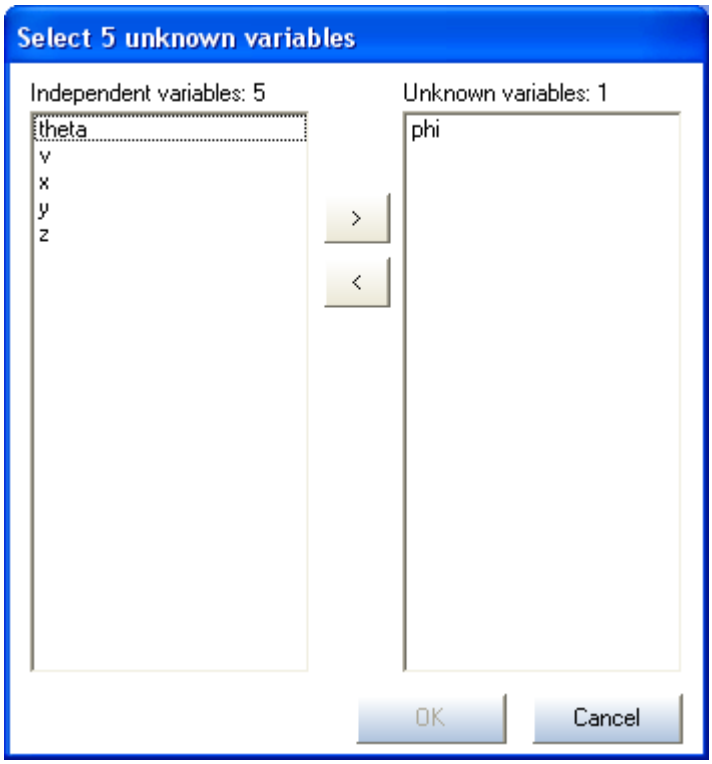


Figure 71 The "Select unknown variables" dialog box

6. In the **Select unknown variables** dialog box, you can either select the required number of unknown variables using the [›] and [‹] buttons and, when done, click **OK**, or click **Cancel** at any time. If you select the required number of unknown variables and click **OK**, the resulting system of equations will appear in the equation system editor as shown in **Figure 72**. Otherwise, if you click **Cancel**, the resulting system of equations will also appear in the equation system editor, but in that case the selection of unknown variables will be deferred to the stage of processing the aggregate system of equations.

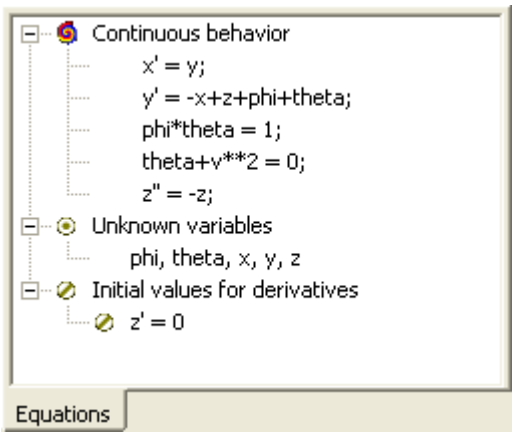


Figure 72 Resulting system of equations, example 2

7. In any case, after you get back to the equation system editor, the selected unknown variables, if any, will appear under the **Unknown variables** node. Additionally, if the set of equations includes second-order differential equations, the **Initial values of derivatives** node will appear in the equation system editor, with sub-nodes representing the initial values of the corresponding first derivatives.

❖ **To override inherited equations:**

1. Open the class editor for a class that inherits the system of equations from another class. Activate the equation system editor, which will look as shown in **Figure 73**.

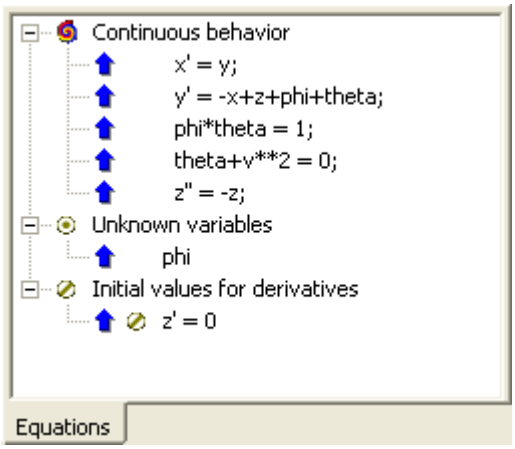


Figure 73 Inherited system of equations

2. Double-click the **Continuous behavior** node or any of its sub-nodes, or Right-click the **Continuous behavior** node or any of its sub-nodes and choose **Edit...** from the popup menu.
The **Text Editor** window will appear as shown in **Figure 74**.

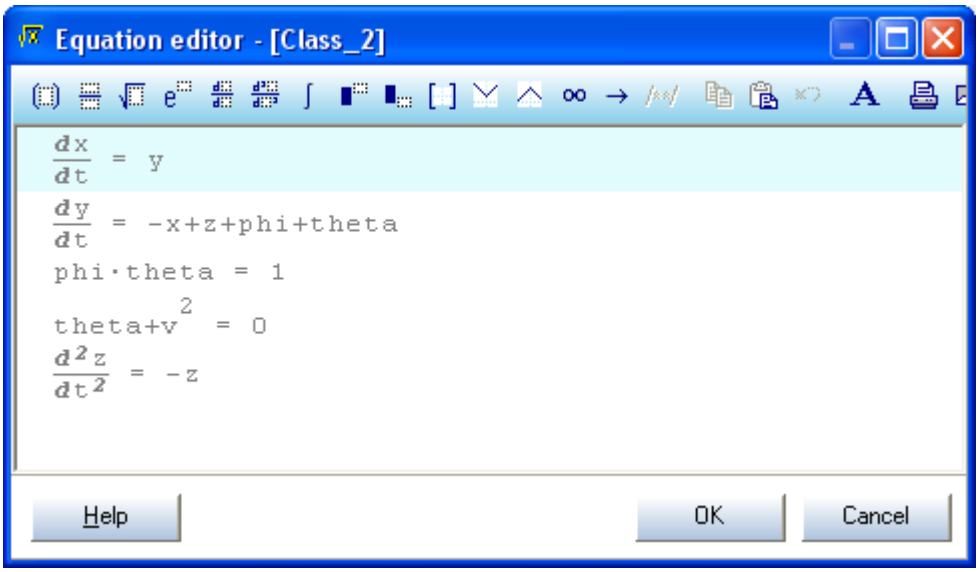


Figure 74 Overriding inherited equations

3. Use the **Text Editor** window to modify the equations.
4. When ready, click **OK**.
   A confirmation message box will be displayed.
5. Click **Yes**.
6. If necessary, select the unknown variables in the **Select unknown variables** dialog box, if it appears.
7. The modified system of equations will appear in the equation system editor. The overridden elements will be displayed with the ◆ sign, as shown in **Figure 75**.

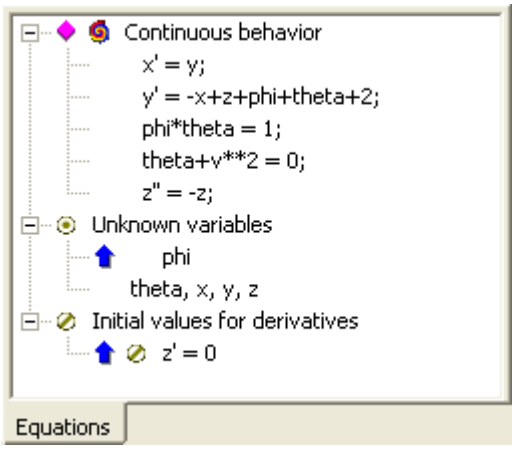


Figure 75 Overridden system of equations

At any moment, the set of unknown variables can be modified via the **Unknown variables** node of the equation system editor.

❖ **To modify the set of unknown variables:**

1. Double-click the **Unknown variables** node or any of its sub-nodes in the equation system editor, or
   Right-click the **Unknown variables** node or any of its sub-nodes in the equation system editor and choose **Edit…** from the popup menu.
   The **Select unknown variables** dialog box will appear as shown in **Figure 73**.

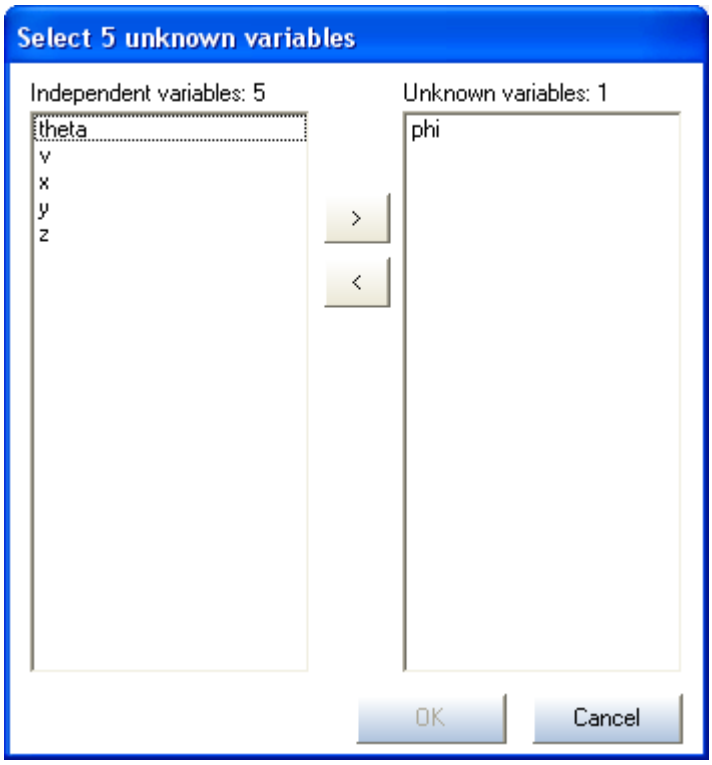


Figure 76 The "Select unknown variables" dialog box

2. Use the [>] and [<] buttons to modify the **Unknown variables** list as necessary.
3. Click **OK**.

If necessary, the initial values of the first derivatives can be modified via the sub-nodes of the **Initial values for derivatives** node.

❖ **To modify the initial value of the first derivative of a variable:**

1. Double-click the corresponding sub-node of the **Initial values for derivatives** node, or
   Right-click the corresponding sub-node of the **Initial values for derivatives** node and choose **Edit…** from the popup menu.
   The **Text Editor** window will appear as shown **Figure 77**.

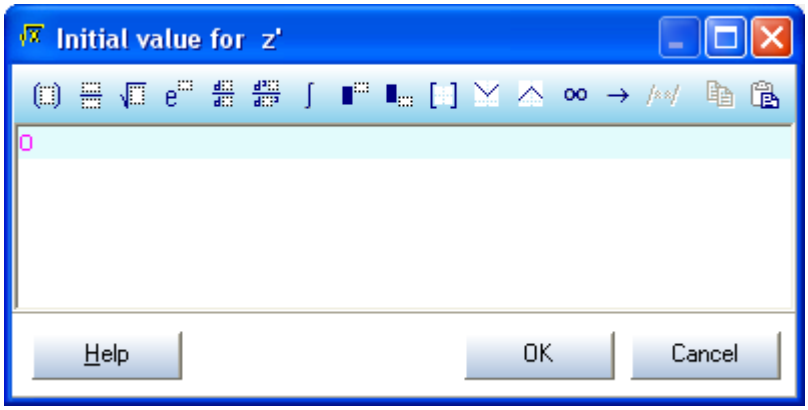


Figure 77 Editing the value of the first derivative of a variable

2. Use the **Text Editor** window to edit the expression for the first derivative.
3. Click **OK**.

3.4.3 Behavior Chart Editor

The behavior chart editor, see **Figure 78**, allows the user to edit the behavior chart of a hybrid class. For more on the behavior chart, see **Section 2.4**.

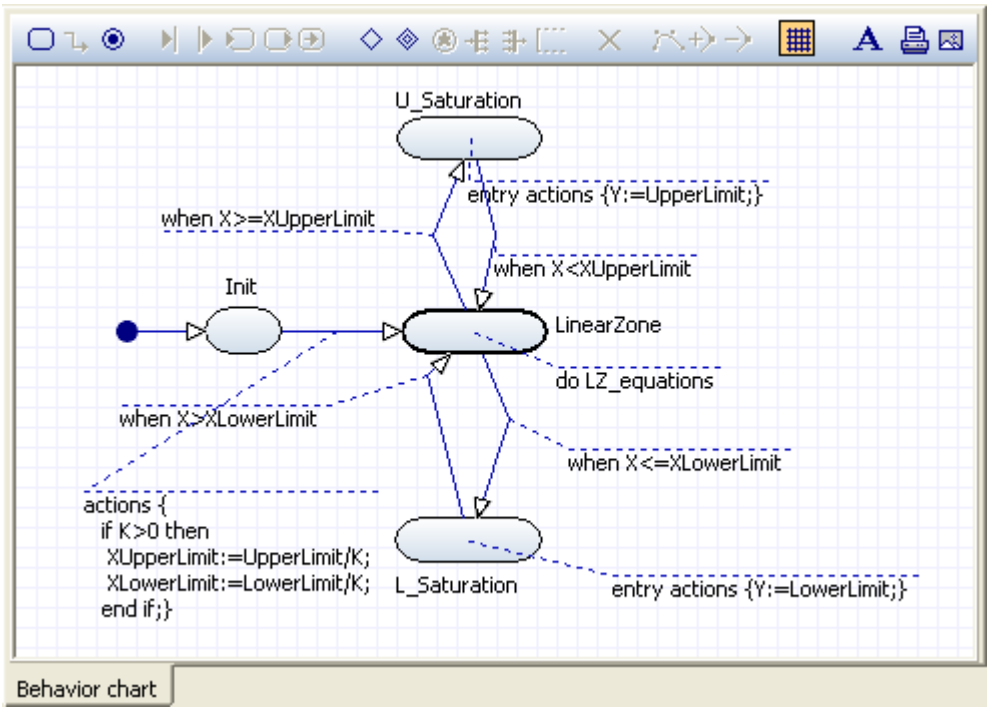


Figure 78 Behavior chart editor

The behavior chart editor consists of the editing area and the toolbar. The editing area allows the user to graphically edit the elements of the behavior chart. The toolbar provides shortcuts to some of the most commonly used commands of the behavior chart editor.

3.4.3.1 Toolbar Buttons

The behavior chart editor toolbar contains the following buttons:

– creates a new state.

– creates a new transition.

– creates a new final state.

– opens the **Transition conditions** dialog box to edit the trigger and guard of the selected transition.

– opens the **Text Editor** window to edit the actions of the selected transition.

– open the **Text Editor** window to edit the entry actions of the selected state.

– opens the **Text Editor** window to edit the exit actions of the selected state.

– opens the **Internal transitions** window to edit the internal transitions of the selected state.

– creates a new switch point.

– creates a new probabilistic switch point.

– deletes the selected elements.

– toggles spline interpolation for the selected transition.

– allows the user to add a base point to the selected transition.

– allows the user to delete a base point from the selected transition.

– toggles the grid in the editing area.

– opens the **Font** dialog box to edit the font used by the behavior chart editor.

– allows the user to print the behavior chart.

– allows the user to save the behavior chart to a file as an image.

3.4.3.2 Editing Area

The behavior chart editing area allows the user to graphically edit the elements of the behavior chart. The editing area can contain the following elements:

States

Special states

Transitions

The user can create new elements, move them around, modify their properties, resize states, modify transition curves, and delete any elements except for the mandatory initial state.

Every element with at least one non-empty property is displayed with an associated property label that shows all non-empty properties of the element. Property labels have links to their respective elements and can be moved around as well.

When a new hybrid class is created, its behavior chart is automatically populated with the mandatory initial state, an ordinary state, and a transition from the initial state to the ordinary one.

❖ **To select an element:**

1. Click the element, or
   Right-click anywhere in the editing area and select the element in the **Go to** submenu of the popup menu.
   The element will appear selected, as shown in **Figure 79**.

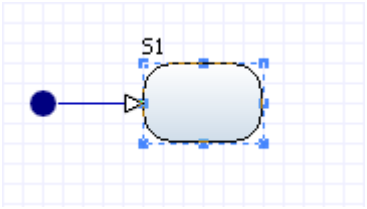


Figure 79 One element selected

❖ **To add an element to the selection:**

1. Click the element while holding *Shift*.
   The element will be added to the selection. This way you can have more than one element selected at a time, see **Figure 80**.

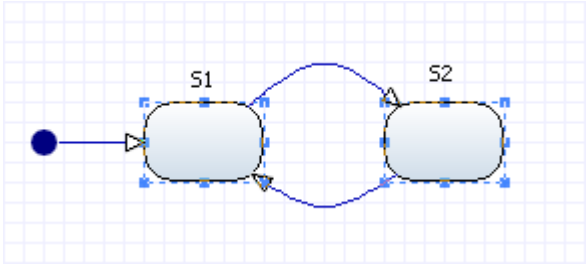


Figure 80 Multiple elements selected

❖ **To select all elements of the behavior chart:**

1. Right-click anywhere in the editing area and choose **Select all** from the popup menu.
   All elements of the behavior chart will be selected, see **Figure 81**.

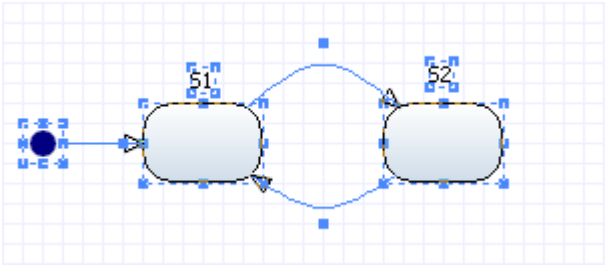


Figure 81 All elements selected

❖ **To move the selected element(s):**

1. Drag the selected element(s) to the desired location.

❖ **To delete the selected element(s):**

1. Click **Delete** on the behavior chart editor toolbar, or
   Right-click the selection and choose **Delete** from the popup menu, or
   Hit *Delete*.

❖ **To toggle the grid:**

1. Click **Grid** on the behavior chart editor toolbar.

Regardless of whether the grid is visible or not, all operations align all elements to the grid.

At any moment, the behavior chart can be converted into a hybrid local class (see **Section 2.11**)

❖ **To convert the behavior chart into a hybrid local class:**

1. Right-click anywhere in the editing area and choose **Behavior chart to class** from the popup menu.
   The **Enter the local class name** dialog box will appear as shown in **Figure 82**.

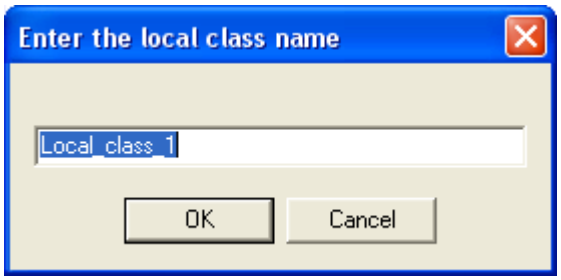


Figure 82 The "Enter the local class name" dialog box

2. Type the name of the new local class and click **OK**.
   This will create a new hybrid local local class with the same behavior chart as the current one, and erase the current behavior chart.

3.4.3.3 States

For more on states, see **Section 2.4.1**.

❖ **To create a new state:**

1. Click **State** on the behavior chart editor toolbar.
   A new state will appear on the behavior chart as shown in **Figure 83**.

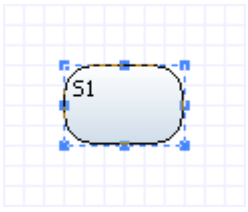


Figure 83 New state

❖ **To resize a state:**

1. Select the state.
2. Drag any border of the state in the desired direction.

❖ **To rename a state:**

1. Double-click the name of the state, or
   Right-click the state and choose **Rename** from the popup menu.
   An in-place editing field will appear as shown in **Figure 84**.

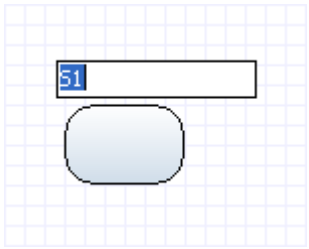


Figure 84 Renaming a state

2. Modify the state name.
3. Hit *Enter* or click anywhere in the editing area.

❖ **To edit the entry actions of a state:**

1. Select the state and click **Entry actions** on the behavior chart editor toolbar, or

Right-click the state and choose **Entry actions…** from the popup menu.
The **Text Editor** window will appear as shown in **Figure 85**.

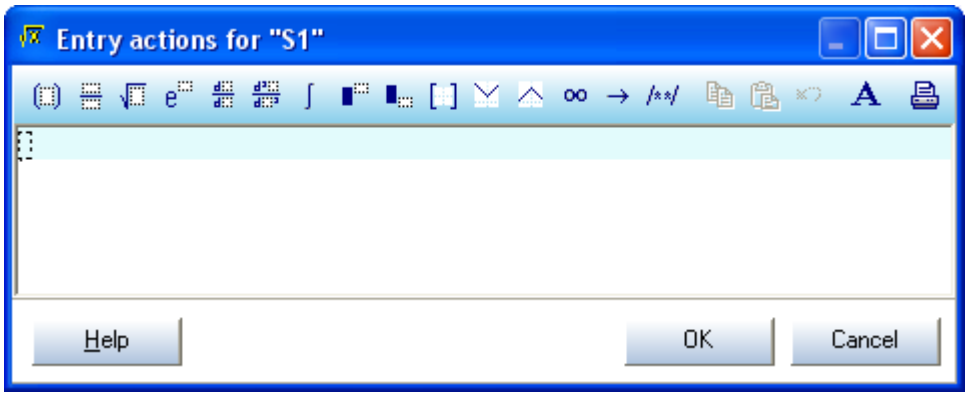


Figure 85 Editing the entry actions of a state

2. Use the **Text Editor** window (see **Section 3.7**) to specify the entry actions of the state.

3. Click **OK** in the **Text Editor** window.
If the entry actions are not empty, they will appear in the property label of the state, as shown in **Figure 86**.

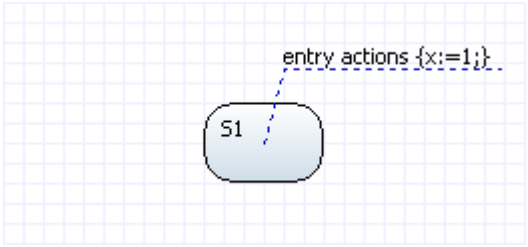


Figure 86 State with entry actions

❖ **To edit the exit actions of a state:**

1. Select the state and click **Exit actions** on the behavior chart editor toolbar, or
Right click the state and choose **Exit actions…** from the popup menu.
The **Text Editor** window will appear as shown in **Figure 87**.

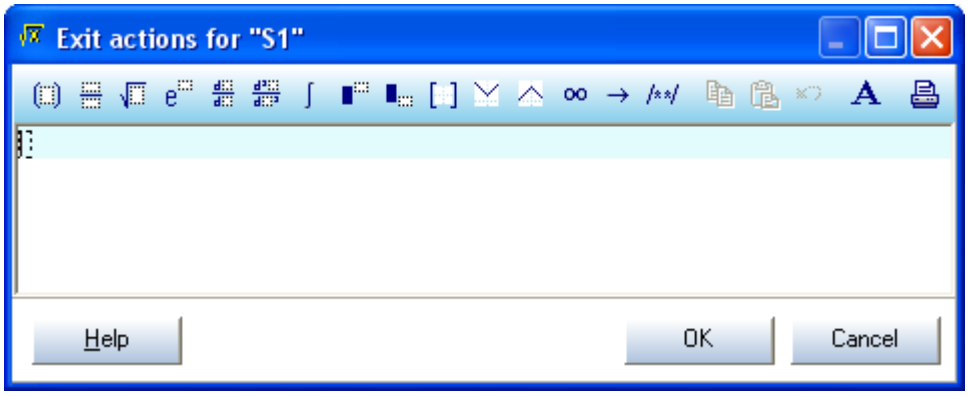


Figure 87 Editing the exit actions of a state

2. Use the **Text Editor** window (see **Section 3.7**) to specify the exit actions of the state.

3. Click **OK** in the **Text Editor** window.
   If the exit actions are not empty, they will appear in the property label of the state, as shown in **Figure 88**.

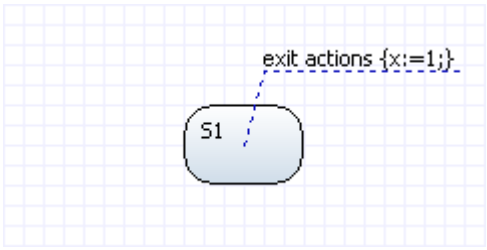


Figure 88 State with exit actions

❖ **To create a continuous local class and associate it with a state:**

1. Right-click the state and choose **Create continuous activity** from the popup menu.
   A new continuous local class will be created and added to the **Local classes | Continuous** node of the class editor tree. The state will be modified to use the created class as the state's activity. The created class will appear in the property label of the state and the state will be displayed with a thick border, as shown in **Figure 89**.

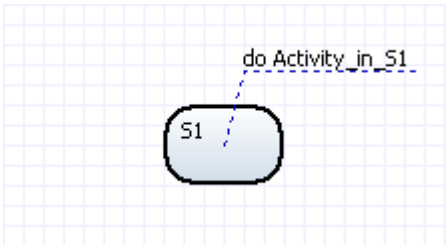


Figure 89 State with continuous activity

❖ **To create a hybrid local class and associate it with a state:**

1. Right-click the state and choose **Create hybrid activity** from the popup menu.
   A new hybrid local class will be created and added to the **Local classes | Hybrid** node of the class editor tree. The state will be modified to use the created class as the state's activity. The created class will appear in the property label of the state and the state will be displayed with a double border, as shown in **Figure 90**.

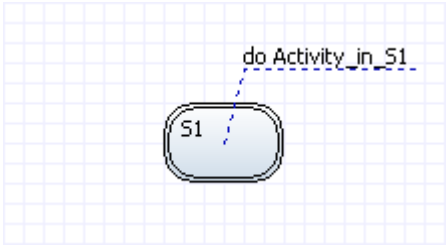


Figure 90 State with hybrid activity

❖ **To use an exiting class as the activity class of a state:**

1. Drag the class from the **Project Manager** window or the class editor tree and drop it onto the state.
   The class will appear in the property label of the state and the state will change its border to reflect the type of the behavior of the class.

❖ **To open the class editor window for the activity class of a state:**

1. Double-click the state, or
   Right-click the state and choose **State activity** from the popup menu.
   The class editor window for the class associated with the state will appear.

❖ **To remove the activity from a state:**

1. Right-click the state and choose **Set empty activity** from the popup menu.
   The activity will disappear from the property label of the state and the state border will change to normal.

❖ **To modify the actual parameter values and initial variable values of a state activity:**

1. Right-click the state and choose **Actual values of parameters and variables…** from the popup menu.
   The **Actual values** dialog box will appear as shown in **Figure 91**.

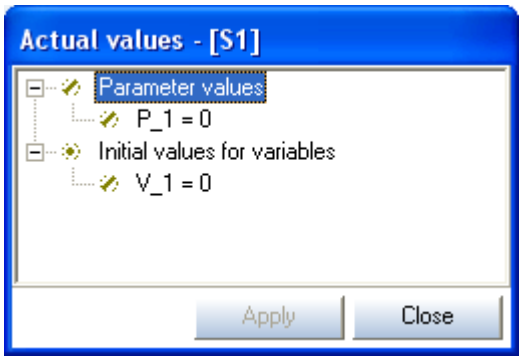


Figure 91 The "Actual values" dialog box

2. To modify the actual value of a parameter or the initial value of a variable, double-click the corresponding item in the **Actual values** dialog box, or right-click the corresponding item in the **Actual values** dialog box and choose **New value…** from the popup menu.
   A value editing dialog box, as shown in **Figure 92**, will appear.

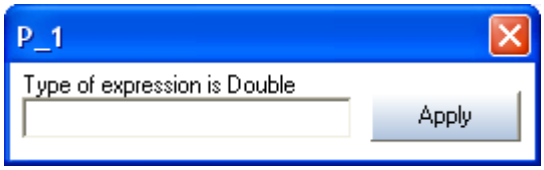


Figure 92 Value editing dialog box

3. Type the expression for the actual parameter value or initial variable value. When done, click **Apply**.
   The **Actual values** dialog box will display the modified parameter or variable with the sign.

4. To restore the actual value of a parameter or the initial value of a variable to the default value, right-click the corresponding item in the **Actual values** dialog box and choose **Restore default value** from the popup menu.

5. Finally, click **Apply** in the **Actual values** dialog box.

❖ **To restore the inherited activity of a state:**

1. Right-click the state and choose **Restore inherited activity** from the popup menu.

3.4.3.4 Transitions

For more on transitions, see **Section 2.4.2**.

❖ **To create a new transition:**

1. Select the source state and click **Transition** on the behavior chart editor toolbar, or
   Right-click the source state and choose **New transition** from the popup menu.
   The behavior chart editor will start displaying a dashed line starting at the border of the source state and ending at the mouse cursor, see **Figure 93**.

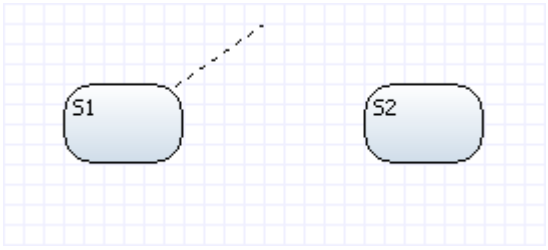


Figure 93 Creating a new transition

2. Click the target state.
   A new transition will appear as shown in **Figure 94**.
   The new transition will go straight from the source state to the target state and will have one intermediate point. If you want the transition to be a more complex curve and/or have more intermediate points, click the locations where the intermediate points should be created before clicking the target state.

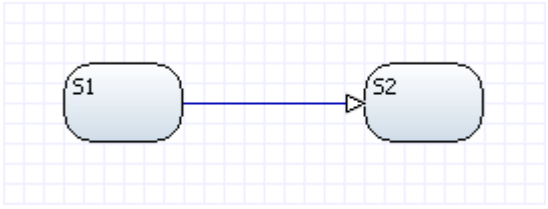


Figure 94 New transition

3. If you want to cancel the operation, click before you click the target state.

The user can modify the curve of any existing transition by adding and removing intermediate points and moving intermediate points.

❖ **To add an intermediate point to a transition:**

1. Select the transition and click **Add control point to transition** on the behavior chart editor toolbar.
2. Move the mouse cursor over the transition and click where you want to add an intermediate point.

❖ **To remove an intermediate point from a transition:**

1. Select the transition and click **Remove control point from transition** on the behavior chart editor toolbar.
2. Click the intermediate point you want to remove.

❖ **To move an intermediate point:**

1. Select the transition and drag the intermediate point in the desired direction.

❖ **To toggle spline interpolation for a transition:**

1. Select the transition and click **Spline curve** on the behavior chart editor toolbar, or
   Right-click the transition and choose **Spline curve** from the popup menu.

❖ **To edit the conditions (trigger, guard, probability) of a transition:**

1. Select the transition and click **Transition conditions** on the behavior chart editor toolbar, or
   Right-click the transition and choose **Transition conditions** from the popup menu.
   The **Transition conditions** dialog box will appear. If the transition exits an ordinary state, the **Transition conditions** dialog box will look as shown in **Figure 95**. If the transition exits a switch point (see **Section 2.4.6**), the **Transition conditions** dialog box will look as shown in **Figure 96**. If the transition exits a probabilistic switch point (see Section **2.4.7**), the **Transition conditions** dialog box will look as shown in **Figure 97**.

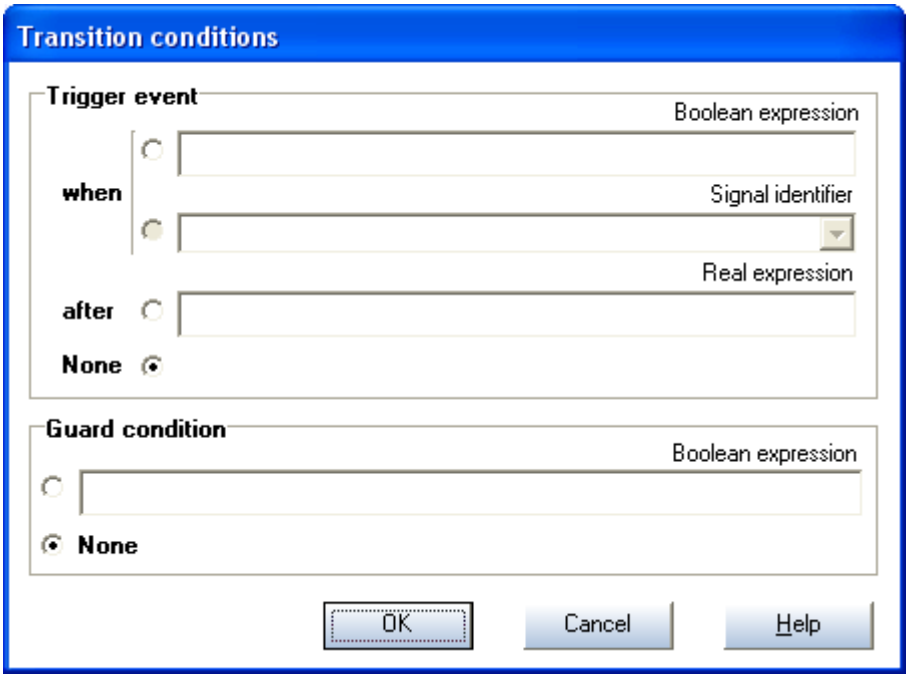


Figure 95 Editing the conditions of a transition exiting an ordinary state

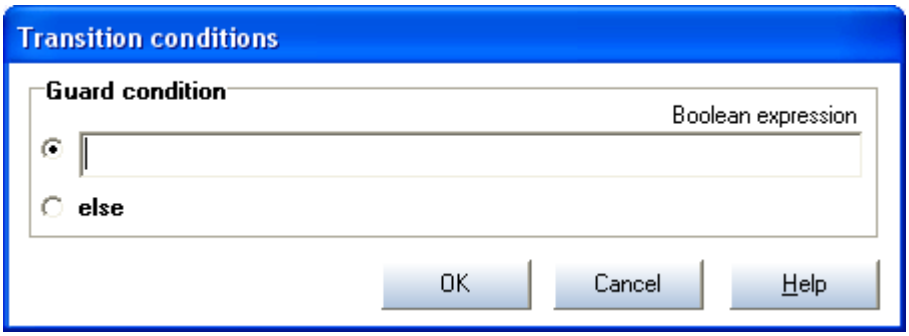


Figure 96 Editing the conditions of a transition exiting a switch point

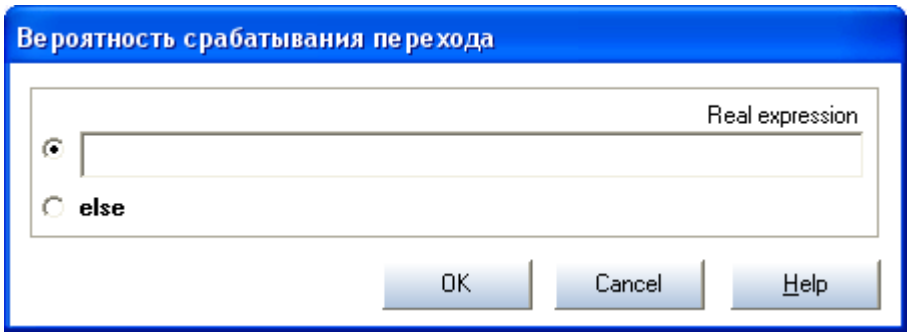


Figure 97 Editing the conditions of a transition exiting a probabilistic switch point

2. Specify the transition conditions.
3. Click **OK**.
   If the conditions are not empty, they will appear in the property label of the transition, as shown in **Figure 98**.

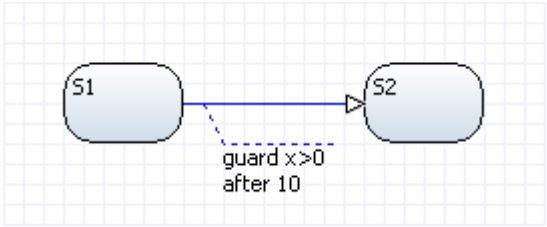


Figure 98 Transition with conditions

❖ **To edit the actions of a transition:**

1. Select the transition and click **Transition actions** on the behavior chart editor toolbar, or
   Right-click the transition and choose **Transition actions...** from the popup menu.
   The **Text Editor** window will appear as shown in **Figure 99**.

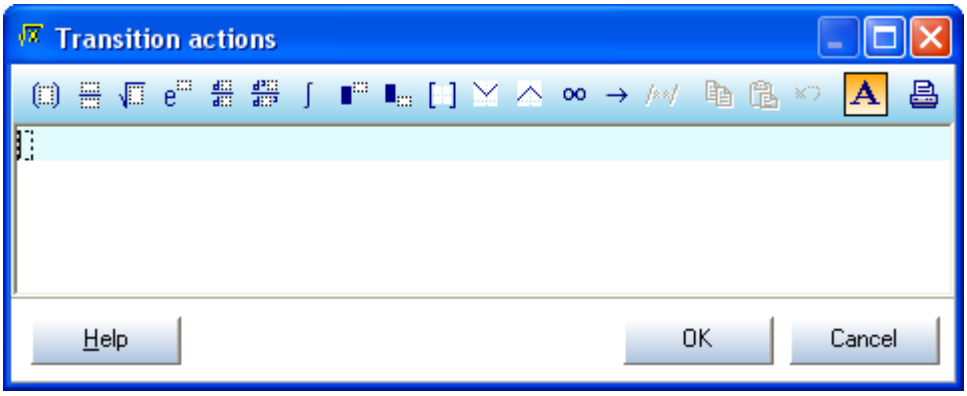


Figure 99 Editing the actions of a transition

2. Use the **Text Editor** window (see **Section 3.7**) to specify the actions for the transition.
3. Click **OK** in the **Text Editor** window.
   If the actions are not empty, they will appear in the property label of the transition, as shown in **Figure 100**.

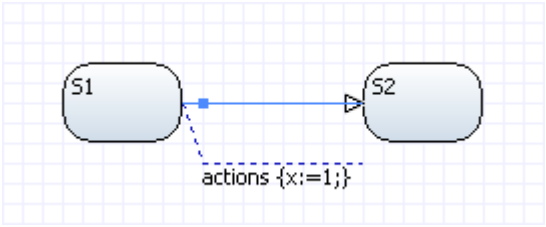


Figure 100 Transition with actions

❖ **To create an internal transition in a state:**

1. Select the state and click **Internal transitions** on the behavior chart editor toolbar, or
   Right-click the state and choose **Internal transitions** from the popup menu.
   The **Internal transitions** window will appear as shown in **Figure 101**.

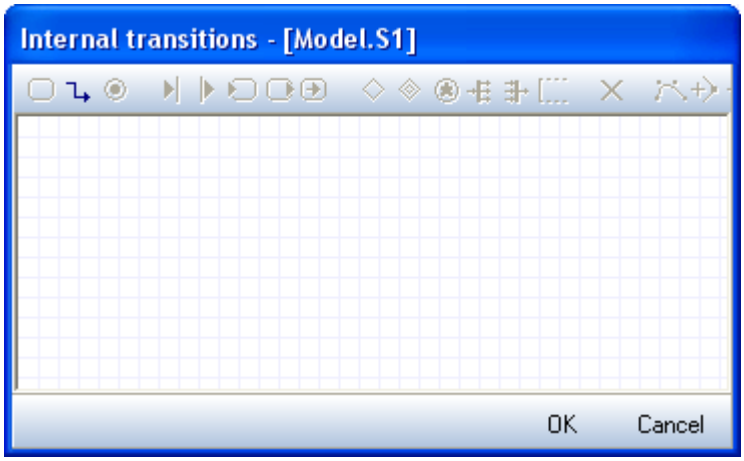


Figure 101 The "Internal transitions" window

2. Click **Transition** on the toolbar of the **Internal transitions** window, or
   Right-click anywhere in the **Internal transitions** window and choose **New transition** from the popup menu.

3. Click anywhere in the **Internal transitions** window.
   A new transition will be created and displayed as shown in **Figure 102**.

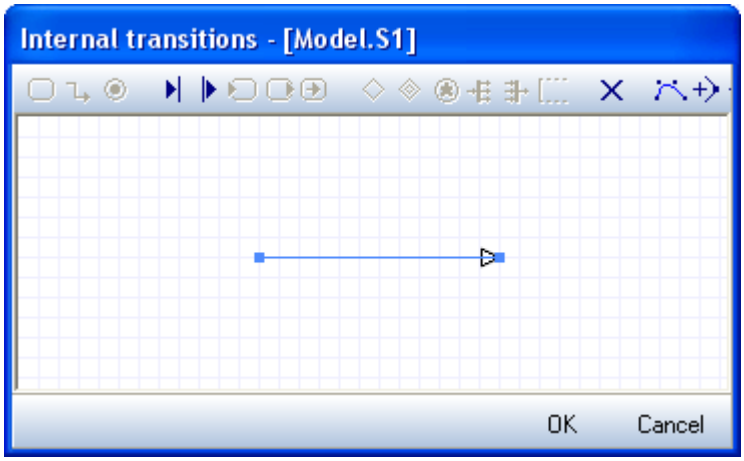


Figure 102 New internal transition

4. Edit the properties of the new transition as if it was an ordinary transition.

5. When done, click **OK** in the **Internal transitions** window.
   The state will now be displayed with an arrow inside, as shown in **Figure 103**, indicating that the state has internal transitions.

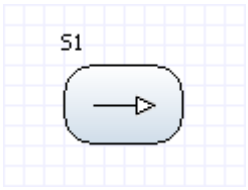


Figure 103 State with internal transitions

❖ **To modify or delete an internal transition:**

1. Select the corresponding state and click **Internal transitions** on the behavior chart editor toolbar, or
   Right-click the state and choose **Internal transitions** from the popup menu.
   The **Internal transitions** window will appear as shown in **Figure 101**.
2. Modify or delete the internal transition as if it was an ordinary transition.
3. Click **OK** in the **Internal transitions** window.

3.4.3.5 Special States

Special states are final state (see **Section 2.4.5**), switch point (see **Section 2.4.6**), and probabilistic switch point (see **Section 2.4.7**).

❖ **To create a final state, a switch point, or a probabilistic switch point:**

1. Click **Final state**, **Switch point**, or **Probabilistic switch point** on the behavior chart editor toolbar.
   A new special state will appear as shown in **Figure 104**, **Figure 105**, or **Figure 106**.

Figure 104 New final state

Figure 105 New switch point

Figure 106 New probabilistic switch point

3.4.4 Structure Diagram Editor

The structure diagram editor, see **Figure 107**, allows the user to edit the structure diagram of a class. For more on the structure diagram, see **Section 2.7**.

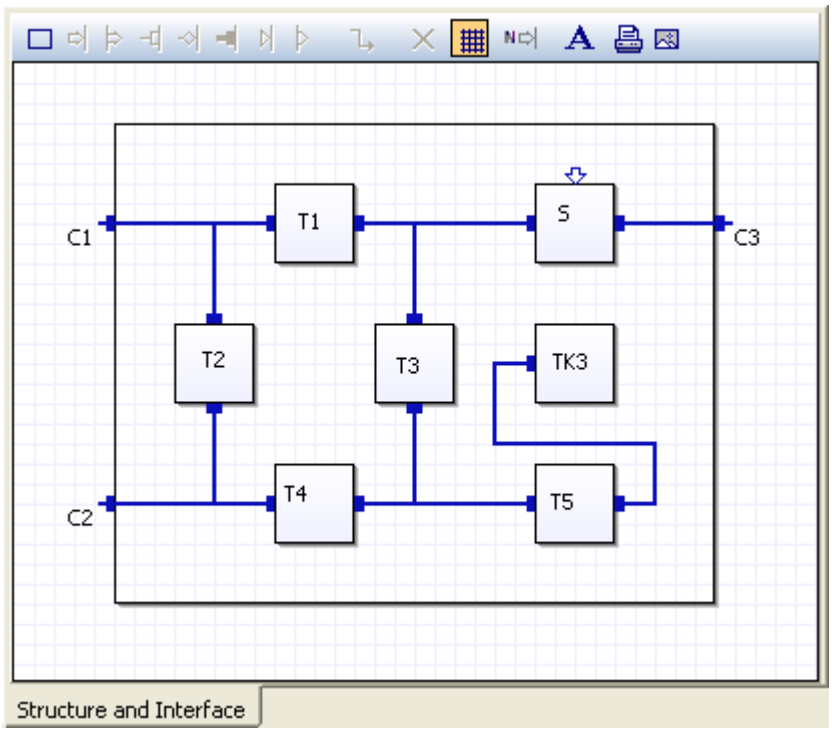


Figure 107 Structure diagram editor

The structure diagram editor consists of the editing area and the toolbar. The editing area allows the user to graphically edit the elements of the structure diagram. The toolbar provides shortcuts to some of the most commonly used commands of the structure diagram editor.

3.4.4.1 Toolbar Buttons

The structure diagram editor toolbar contains the following buttons:

– creates a new local object.

– opens the **External variable** dialog box to create an input variable.

– opens the **External variable** dialog box to create an output variable.

– opens the **External variable** dialog box to create a contact variable.

– opens the **External variable** dialog box to create a flow variable.

– opens the **External variable** dialog box to create a connector variable.

– creates a new link.

– deletes the selected elements.

– toggles the grid in the editing area.

– toggles display of local object variable names.

– opens the **Font** dialog box to edit the font used by the structure diagram editor.

– allows the user to print the structure diagram.

– allows the user to save the structure diagram to a file as an image.

3.4.4.2 Editing Area

The structure diagram editing area allows the user to graphically edit the elements of the structure diagram. The editing area can contain the following elements:

Local objects (if the class is compound, see **Section 2.7.1**)

Class boundary (if the class is open, see **Section 2.7.2**)

External variables (see **Section 2.9**)

Links (if the class is compound, see **Section 2.7.3**)

The user can create elements and, in most cases, move them around, resize them, modify their properties, and delete them. Local objects and external variables have associated names that can be moved around as well. Local objects can be rotated.

The class boundary represents the interface of the class and cannot be created or deleted. Instead, it appears and disappears automatically depending on whether the class is open or isolated.

External variables of the class are displayed on the class boundary and can be moved around the boundary but cannot be detached from it. External variables of local objects represent external variables defined in respective classes and cannot be moved, created, or deleted when editing the structure diagram of the containing class.

Most elements have tooltips that display their properties.

❖ **To select an element:**

1. Click the element, or
   Right-click anywhere in the editing area and select the element in the **Go to** submenu of the popup menu.
   The element will appear selected, as shown in **Figure 108**.

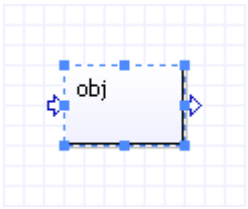


Figure 108 One element selected

❖ **To add an element to the selection:**

1. Click the element while holding *Shift*.
   The element will be added to the selection. This way you can have more than one element selected at a time, see **Figure 109**.

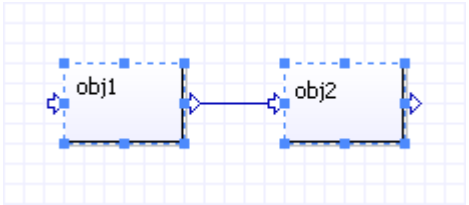


Figure 109 Multiple elements selected

❖ **To select all elements of the structure diagram:**

1. Right click anywhere in the editing area and choose **Select all** from the popup menu.
   All elements of the structure diagram will be selected, see **Figure 110**.

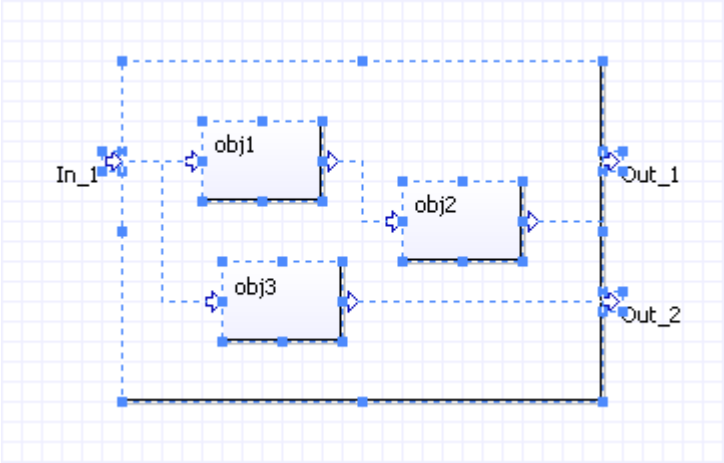


Figure 110 All elements selected

❖ **To move the selected element(s):**

1. Drag the selected element(s) to the desired location.

❖ **To delete the selected element(s):**

1. Click **Delete** on the structure diagram editor toolbar, or
   Right-click the selection and choose **Delete** from the popup menu, or
   Hit *Delete*.

❖ **To toggle the grid:**

1. Click **Grid** on the structure diagram editor toolbar.

Regardless of whether the grid is visible or not, all operations align all elements to the grid.

3.4.4.3 Local Objects

For more on local objects, see **Section 2.7.1**.

A new local object can be created by either dragging a class from the **Project Manager** window and dropping it onto the structure diagram, or using the **Add object** toolbar button of the structure diagram editor.

❖ **To create a new local object via drag-and-drop:**

1. Drag a class from the **Project Manager** window and drop it onto the structure diagram editing area.
   A new local object of the corresponding class will appear, as shown in **Figure 111**.

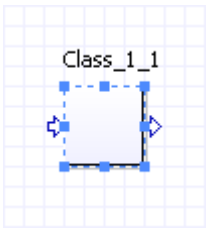


Figure 111 New local object

❖ **To create a new local object using the "Add object" toolbar button:**

1. Click **Add object** on the structure diagram editor toolbar.
   The **Choose class** dialog box will appear as shown in **Figure 112**.

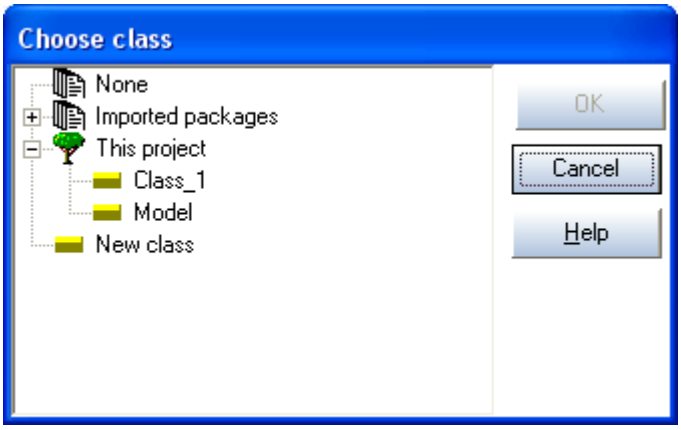


Figure 112 The "Choose class" dialog box

2. Select the class from which you want to create a local object.
3. Click **OK**.
   A new local object of the selected class will appear as shown in **Figure 111**.

❖ **To resize a local object:**

1. Select the local object.
2. Drag any border of the local object in the desired direction.

❖ **To rotate a local object:**

1. Right-click the local object and choose **Rotate | 90° clockwise** , **Rotate | 90° counterclockwise** , or **Rotate | 180°** from the popup menu.

❖ **To rename a local object:**

1. Double-click the name of the local object, or
   Right-click the local object and choose **Rename** from the popup menu.
   An in-place editing field will appear as shown in **Figure 113**.

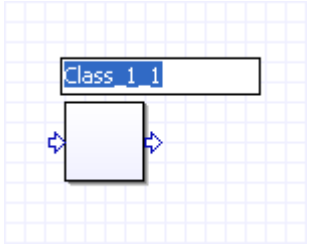


Figure 113 Renaming a local object

2. Modify the local object name.
3. Hit *Enter* or click anywhere in the editing area.

❖ **To edit the comments of a local object:**

1. Right-click the local object and choose **Edit comments** from the popup menu.
   An in-place editing field will appear as shown in **Figure 114**.

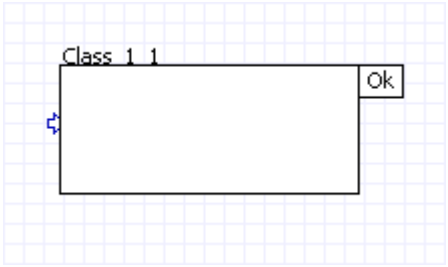


Figure 114 Editing the comments of a local object

2. Modify the local object comments.
3. Click **OK**.

❖ **To edit the actual parameter values and initial variable values of a local object:**

1. Right-click the local object and choose
   Actual values of parameters and variables… from the popup menu.
   The **Actual values** dialog box will appear as shown in **Figure 115**.

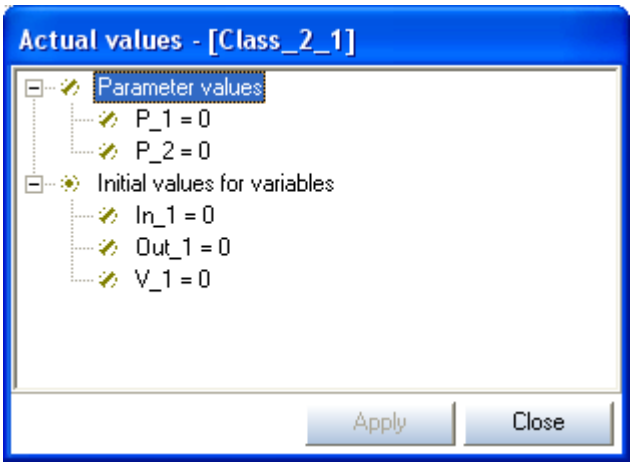


Figure 115 The "Actual values" dialog box

2. To modify the actual value of a parameter or the initial value of a variable, double-click the corresponding item in the **Actual values** dialog box, or right-click the corresponding item in the **Actual values** dialog box and choose **New value…** from the popup menu.
   A value editing dialog box, as shown in **Figure 116**, will appear.

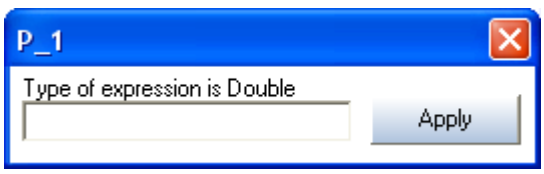


Figure 116 Value editing dialog box

3. Type the expression for the actual parameter value or initial variable value. When done, click **Apply**.
   The **Actual values** dialog box will display the modified parameter or variable with the sign.

4. To restore the actual value of a parameter or the initial value of a variable to the default value, right-click the corresponding item in the **Actual values** dialog box and choose **Restore default value** from the popup menu.

5. Finally, click **Apply** in the **Actual values** dialog box.

❖ **To open the class editor window for the class of a local object:**

1. Double-click the local object.

3.4.4.4 Class Boundary

The class boundary, see **Section 2.7.2**, represents the interface of the class and cannot be created or deleted. Instead, it appears and disappears automatically depending on whether the class is open or isolated. However, the class boundary can be resized.

❖ **To resize the class boundary:**

1. Select the class boundary.

2. Drag any border of the boundary in the desired direction.

3.4.4.5 External Variables

External variables shown on a structure diagram can belong to either the class or local objects. External variables of the class are displayed on the class boundary and can be created, deleted, modified, and moved around the boundary. External variables of local objects represent external variables defined in respective classes and cannot be moved, created, modified, or deleted when editing the structure diagram of the containing class. For more on external variables, see **Section 2.9**.

❖ **To create a new external variable:**

1. Select the class boundary.

2. Click **Add input**, **Add output**, **Add contact**, **Add flow**, or **Add connector** on the structure diagram editor toolbar.
   The **External variable** dialog box will appear as shown in **Figure 117**.

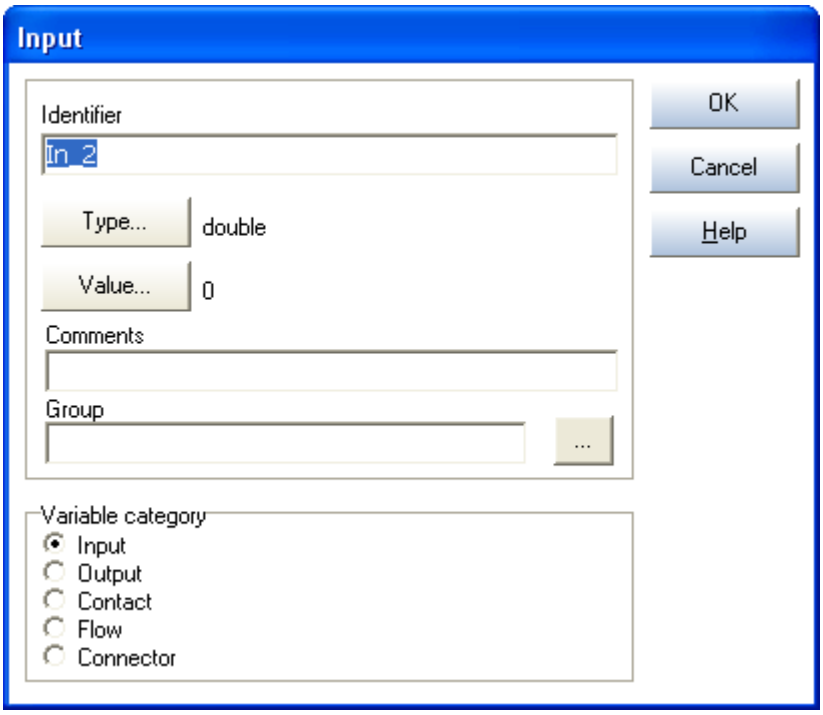


Figure 117 The "External variable" dialog box

3. Use the **External variable** dialog box to define an external variable.
4. Click **OK**.
   The new external variable will appear on the border of the class, as shown in **Figure 118**.

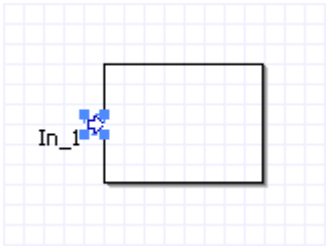


Figure 118 New external variable

❖ **To modify an external variable:**

1. Double-click the external variable.
   The **External variable** dialog box will appear as shown in **Figure 117**.
2. Use the **External variable** dialog box to modify the variable.
3. Click **OK**.

❖ **To rename an external variable in-place:**

1. Double-click the name of the variable, or
   Right-click the name of the variable and choose **Rename** from the popup menu.
   An in-place editing field will appear as shown in **Figure 119**.

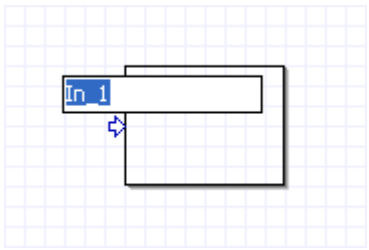


Figure 119 Renaming an external variable in-place

2. Modify the variable name.
3. Hit *Enter* or click anywhere in the editing area.

By default, the structure diagram editor does not display local object variable names.

❖ **To toggle display of local object variable names:**

1. Click **Show local object variable names** on the structure diagram editor toolbar.

3.4.4.6 Links

For more on links, see **Section 2.7.3**.

❖ **To create a new link:**

1. Select an external variable and click **Add link** on the structure diagram editor toolbar, or
   Right-click an external variable and choose **Add new link** from the popup menu.
   The structure diagram editor will start displaying a dashed polyline between the selected variable and mouse cursor, see **Figure 120**.

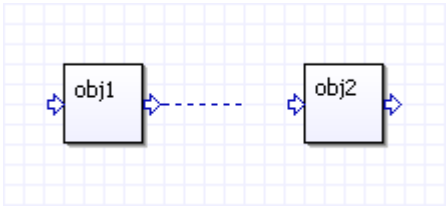


Figure 120 Creating a new link

2. Click another external variable.
   A new link will appear connecting the two variables, as shown in **Figure 121**.
   The new link will have the minimum required number of intermediate points. If you want the link to have more intermediate points, click the locations where the intermediate points should be created before clicking the second variable.

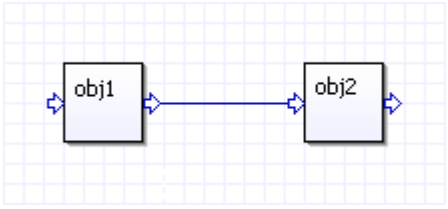


Figure 121 New link

3. If you want to cancel the operation, click or hit *Esc* before you click the second variable.

❖ **To modify a link:**

1. Click on a segment of the link.
2. Drag the segment to the desired location. Vertical segments can only be moved horizontally. Horizontal segments can only be moved vertically.

3.4.5 The Model Class Features

Every project contains a special class called `Model`. The `Model` class is special in that it is interpreted by Rand Model Designer as the complete definition of the model. Rand Model Designer automatically creates an instance of the `Model` class – the *model object* – upon executable model startup and uses it to drive the model execution. The `Model` class exists in every project and cannot be deleted. For more on the `Model` class, see **Section 2.1**.

The integrated modeling environment provides a set of functions specific to the `Model` class, allowing the user to:

Move the contents of the `Model` class into another class

Move the contents of another class into the `Model` class

Erase the contents of the `Model` class

Edit the actual parameter values and initial variables values of the model object

These functions are available from within the `Model` class editor as well as through the main menu and toolbar of the integrated modeling environment.

❖ **To move the contents of the Model class into another class:**

1. Click **Save model as class** on the main toolbar, or
   Choose **Model | Save model as class** from the main menu, or
   Right-click anywhere in the class editor tree of the `Model` class and choose **Save model as class** from the popup menu.
   The **Enter the new class name** dialog box will appear as shown in **Figure 122**.

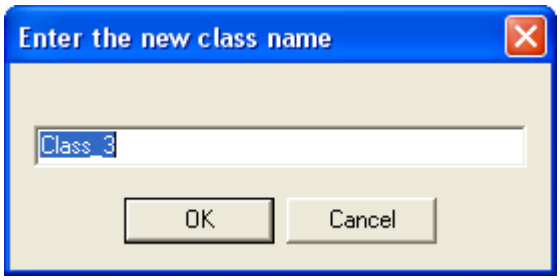


Figure 122 The "Enter the new class name" dialog box

2. Type a new class name.
3. Click **OK**.
   A new class with the specified name with be created. The contents of the `Model` class will be moved into the new class, that is, copied into the new class and erased in the `Model` class.

❖ **To move the contents of another class into the Model class:**

1. Right-click anywhere in the class editor tree of the `Model` class and choose **Load model from class** from the popup menu.
   The **Choose class** dialog box will appear as shown in **Figure 123**.

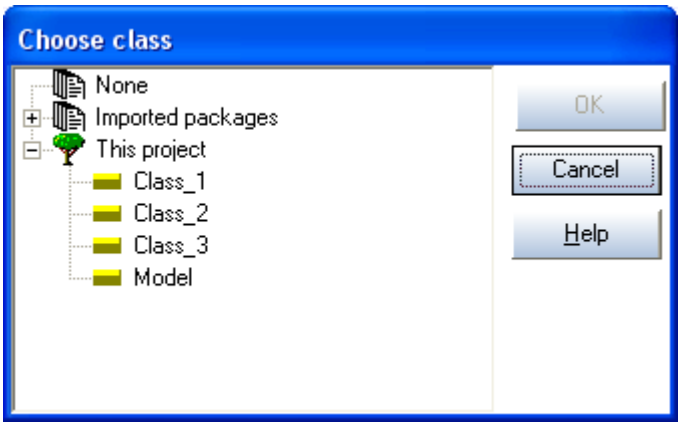


Figure 123 The "Choose class" dialog box

2. Select the class whose contents is to be moved into the `Model` class.

3. Click **OK**.
   The contents of the selected class will be moved into the `Model` class. The selected class will be deleted.

❖ **To erase the contents of the Model class:**

1. Click **Clear model description** on the main toolbar, or
   Choose **Model | Clear model description** from the main menu, or
   Right-click anywhere in the class editor tree of the `Model` class and choose **Clear model description** from the popup menu.
   A confirmation message box will be displayed.

2. Click **Yes**.

❖ **To edit the actual parameter values and initial variable values of the model object:**

1. Click **Actual model parameters** on the main toolbar, or
   Choose **Model | Actual model parameters…** from the main menu, or
   Right-click anywhere in the class editor tree of the `Model` class and choose **Actual model parameters…** from the popup menu.
   The **Actual values** dialog box will appear as shown in **Figure 124**.

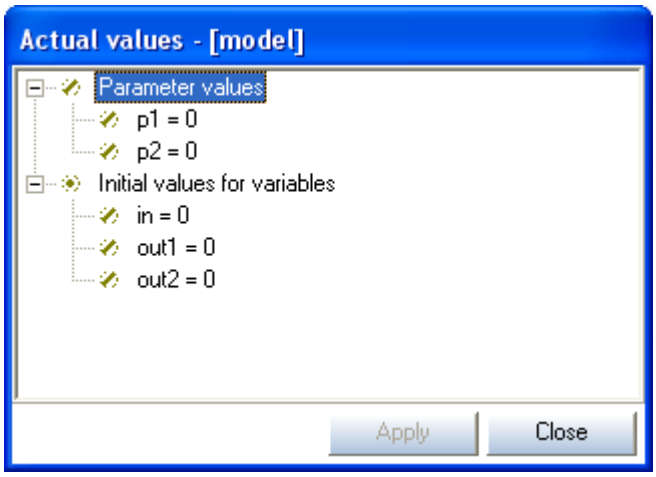


Figure 124 The "Actual values" dialog box

2. To modify the actual value of a parameter or the initial value of a variable, double click the corresponding item in the **Actual values** dialog box, or right-click the corresponding item in the **Actual values** dialog box and choose

New value… from the popup menu.
A value editing dialog box, as shown in **Figure 125**, will appear.

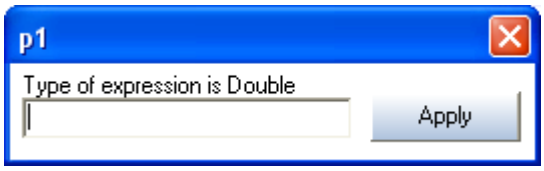


Figure 125 Value editing dialog box

3. Type the expression for the actual parameter value or initial variable value. When done, click **Apply**.
   The **Actual values** dialog box will display the modified parameter or variable with the sign.
4. To restore the actual value of a parameter or the initial value of a variable to the default value, right-click the corresponding item in the **Actual values** dialog box and choose **Restore default value** from the popup menu.
5. Finally, click **Apply** in the **Actual values** dialog box.

3.5 THE "CHOOSE TYPE" DIALOG BOX

The **Choose type** dialog box, see **Figure 126**, allows the user to specify a type.

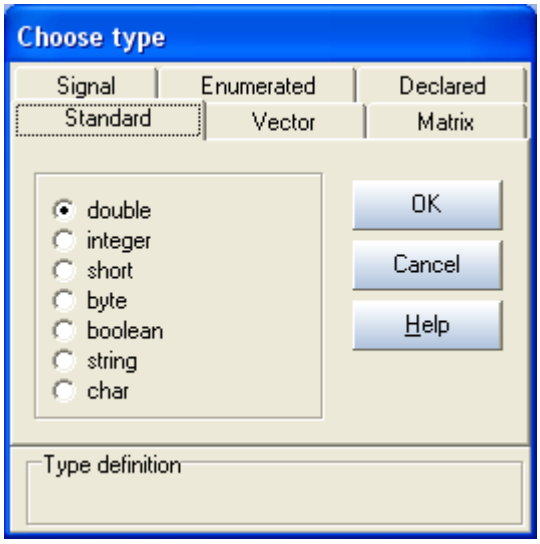


Figure 126 The "Choose type" dialog box, the "Standard" tab

The **Choose type** dialog box contains several tabs, each dedicated to a certain kind of types. The set of tabs available in every particular situation depends on the context in which the **Choose type** dialog box is used. In general, the dialog box allows specifying any type supported by Rand Model Designer. For more on types, see **Section 2.13**.

❖ **To specify a scalar type:**

1. Click the **Standard** tab, if it is not active.
   The **Choose type** dialog box will look as shown in **Figure 126**.
2. Check the required type.
3. Click **OK**.

❖ **To specify a vector type:**

1. Click the **Vector** tab.
   The **Choose type** dialog box will look as shown in **Figure 127**.

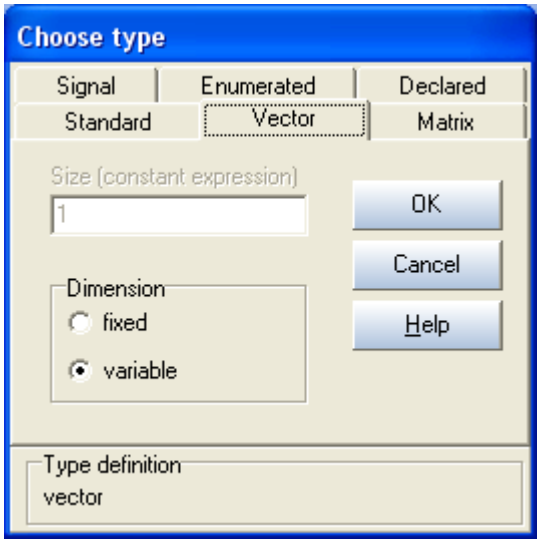


Figure 127 The "Choose type" dialog box, the "Vector" tab

2. In the **Dimension** section, check **fixed** for a fixed-size vector type, or **variable** for a variable-size vector type.
3. If you chose a fixed-size vector type, type the vector size in the **Size** field.
4. Click **OK**.

❖ **To specify a matrix type:**

1. Click the **Matrix** tab.
   The **Choose type** dialog box will look as shown in **Figure 128**.

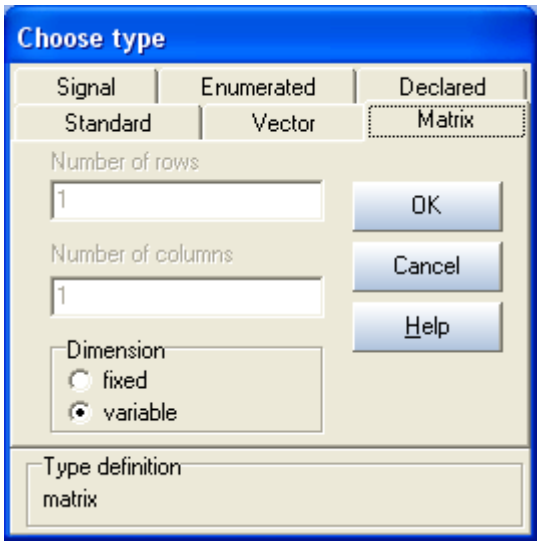


Figure 128 The "Choose type" dialog box, the "Matrix" tab

2. In the **Dimension** section, check **fixed** for a fixed-size matrix type or **variable** for a variable-size matrix type.
3. If you chose a fixed-size matrix type, type the number of rows in the **Number of rows** field and the number of columns in the **Number of columns** field.

4. Click **OK**.

❖ **To specify a signal type:**

1. Click the **Signal** tab.
   The **Choose type** dialog box will look as shown in **Figure 129**.

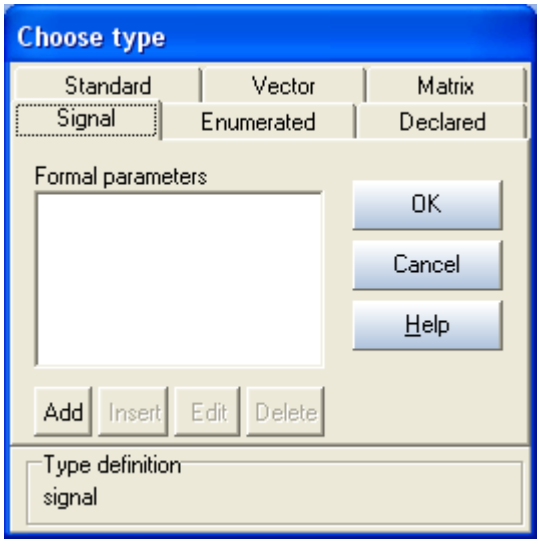


Figure 129 The "Choose type" dialog box, the "Signal" tab

2. To add a parameter to the signal type, click **Add**.
   The **Parameter** dialog box will appear as shown in **Figure 130**.

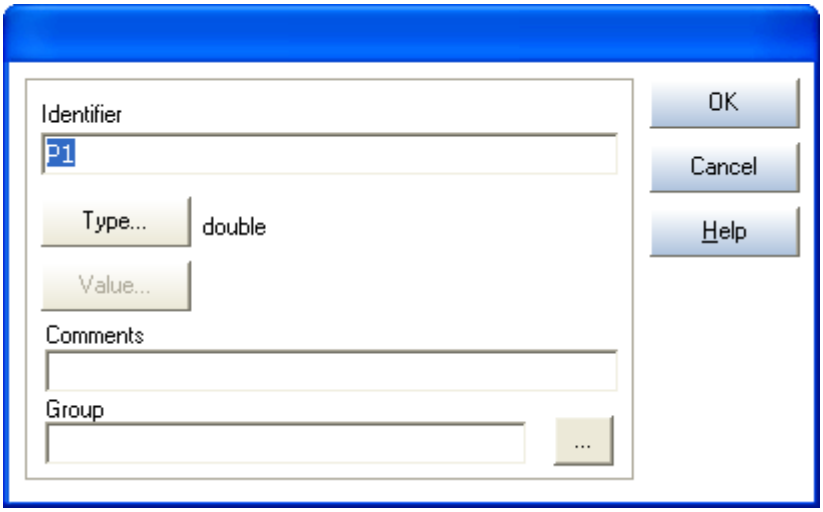


Figure 130 The "Parameter" dialog box

3. Type the name of the parameter in the **Identifier** field.

4. To specify the type of the parameter, click **Type…**.
   Another **Choose type** dialog box will appear.
   Use the **Choose type** dialog box to specify the type of the parameter. When done, click **OK** in the **Choose type** dialog box.

5. If necessary, type the comments for the parameter in the **Comments** field.

6. If you want to assign the parameter to one or more groups, click .
   The **Groups** dialog box will appear.
   Use the **Groups** dialog box (see **Section 3.6**) to assign the parameter to one or more groups. When done, click **Apply** in the **Groups** dialog box.

7. Click **OK** in the **Parameter** dialog box.
8. If necessary, repeat the above steps to add more parameters to the signal type.
9. Click **OK** in the **Choose type** dialog box.

❖ **To specify an enumeration:**

1. Click the **Enumerated** tab.
   The **Choose type** dialog box will look as shown in **Figure 131**.

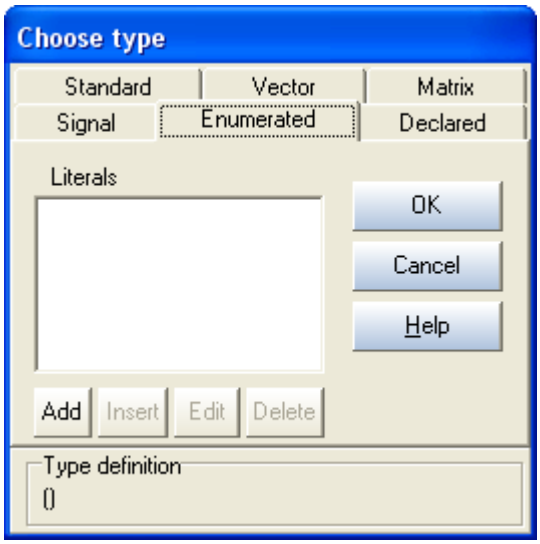


Figure 131 The "Choose type" dialog box, the "Enumerated" tab

2. To add a literal to the enumeration, click **Add**.
   The **Enter the enumerated literal** dialog box will appear as shown in **Figure 132**.

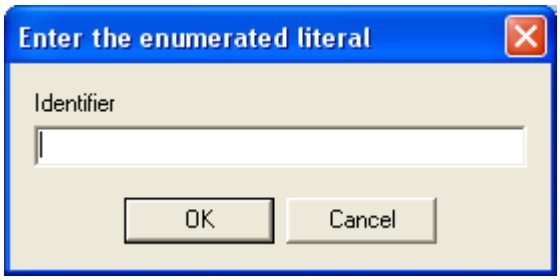


Figure 132 The "Enter the enumerated literal" dialog box

3. Type the literal in the **Identifier** field and click **OK**.
4. Repeat the last two steps to add more literals to the enumeration.
5. Click **OK** in the **Choose type** dialog box.

❖ **To specify a declared type:**

1. Click the **Declared** tab.
   The **Choose type** dialog box will look as shown in **Figure 133**.

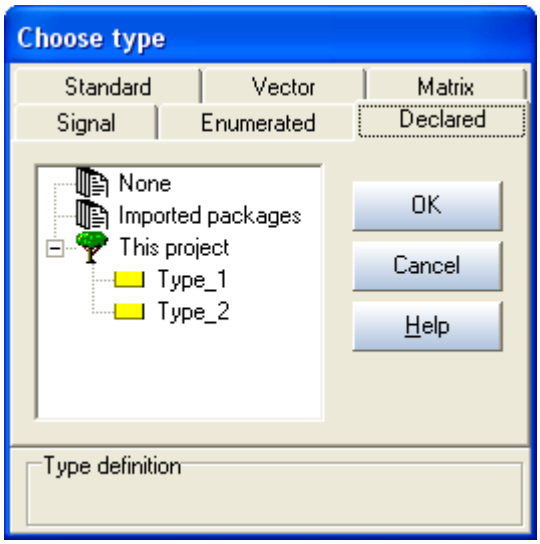


Figure 133 The "Choose type" dialog box, the "Declared" tab

2. Select the required type and click **OK**.

When editing the type for which a declared type acts as a synonym, two additional categories of types become available: record types and connecter types.

❖ **To specify a record type:**

1. Click the **Record** tab.
   The **Choose type** dialog box will look as shown in **Figure 134**.

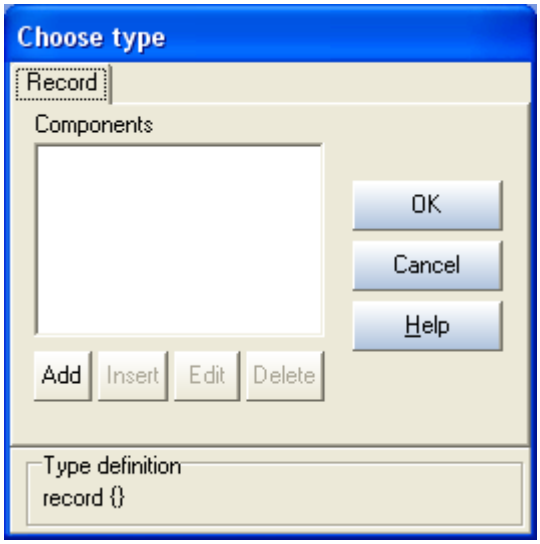


Figure 134 The "Choose type" dialog box, the "Record" tab

2. To add a field to the record type, click **Add**.
   The **Record component** dialog box will appear as shown in **Figure 135**.

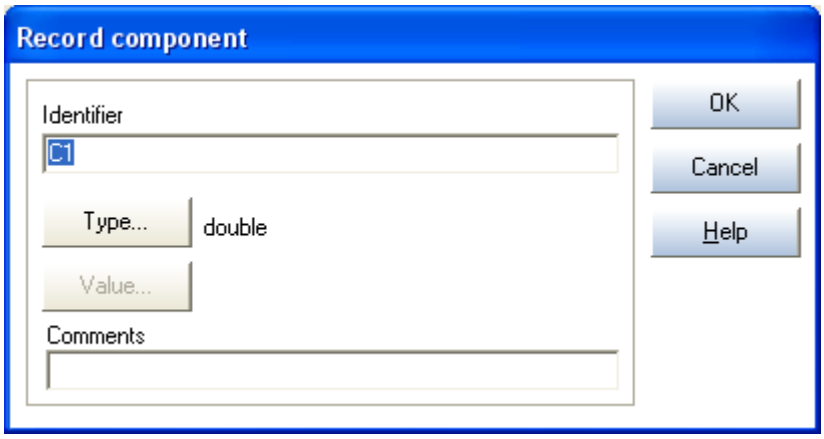


Figure 135 The "Record component" dialog box

3. Type the name of the field in the **Identifier** field.
4. To specify the type of the field, click **Type…**.
   Another **Choose type** dialog box will appear.
   Use the **Choose type** dialog box to specify the type of the field. When done, click **OK** in the **Choose type** dialog box.
5. If necessary, type the comments for the field in the **Comments** field.
6. Click **OK** in the **Record component** dialog box.
7. If necessary, repeat the above steps to add more fields to the record type.
8. Click **OK** in the **Choose type** dialog box.

❖ **To specify a connector type:**

1. Click the **Connector** tab.
   The **Choose type** dialog box will look as shown in **Figure 136**.

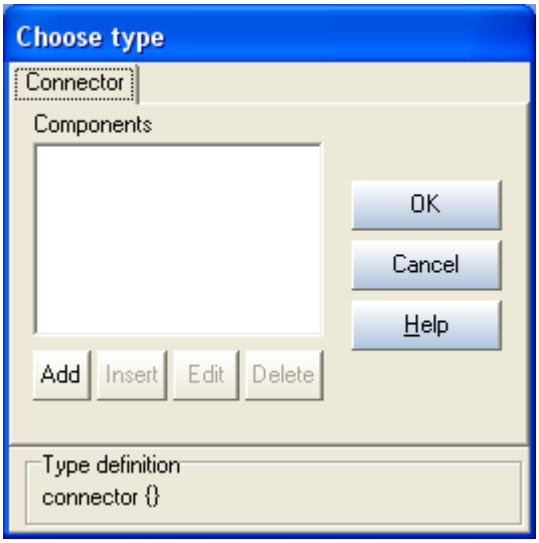


Figure 136 The "Choose type" dialog box, the "Connector" tab

2. To add a component to the connector type, click **Add**.
   The **Connector component** dialog box will appear as shown in **Figure 137**.

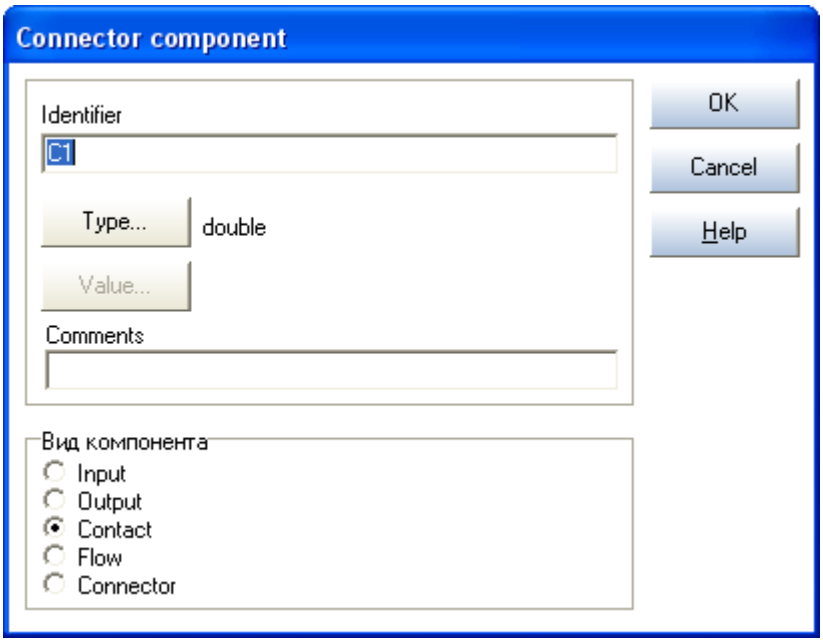


Figure 137 The "Connector component" dialog box

3. Type the name of the component in the **Identifier** field.
4. To specify the type of the component, click **Type…**.
   Another **Choose type** dialog box will appear.
   Use the **Choose type** dialog box to specify the type of the component. When done, click **OK** in the **Choose type** dialog box.
5. If necessary, type the comments for the component in the **Comments** field.
6. Choose the category of the component in the **Component category** section.
7. Click **OK** in the **Connector component** dialog box.
8. If necessary, repeat the above steps to add more components to the connector type.
9. Click **OK** in the **Choose type** dialog box.

3.6 THE "GROUPS" DIALOG BOX

The **Groups** dialog box allows the user to edit the set of groups associated with a parameter, variable, or constant. The **Groups** dialog box looks as shown in **Figure 138**.

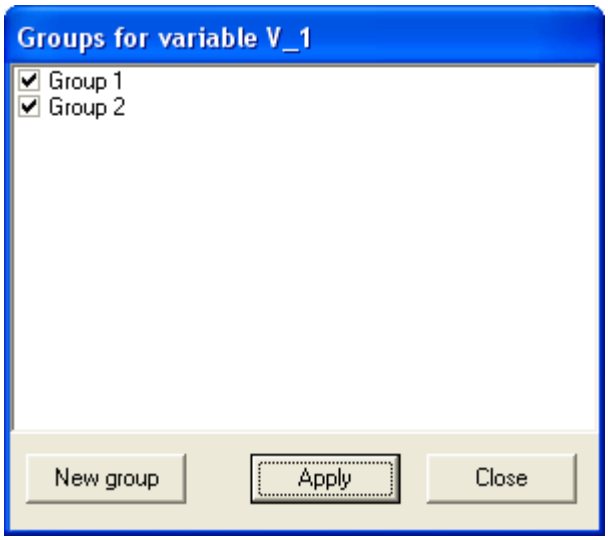


Figure 138 The "Groups" dialog box

❖ **To create a new group:**

1. Click **New group** in the **Groups** dialog box.
   The **Enter the new group name** dialog box will appear as shown in **Figure 139**.

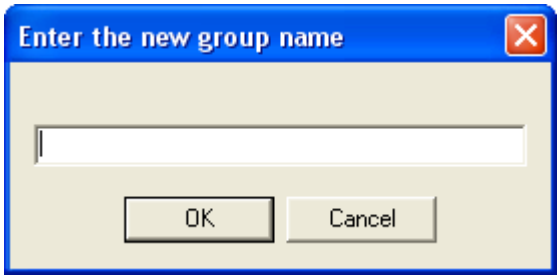


Figure 139 The "Enter the new group name" dialog box

2. Type the name of the new group.
3. Click **OK**.

❖ **To modify the set of groups associated with a parameter, variable, or constant:**

1. Check the groups that should be associated with the parameter variable, or constant.
2. Uncheck the groups that should not be associated with the parameter, variable, or constant.
3. Click **Apply**.

3.7 TEXT EDITOR

The **Text Editor** window allows the user to edit various textual properties including:

Equations (see **Section 2.5**)

Expressions specifying parameter, variable, and constant values (see **Section 2.15**)

Function and procedure code (see **Section 2.14.2**)

State and transition actions (see **Section 2.4.1** and **Section 2.4.2**)

Comments

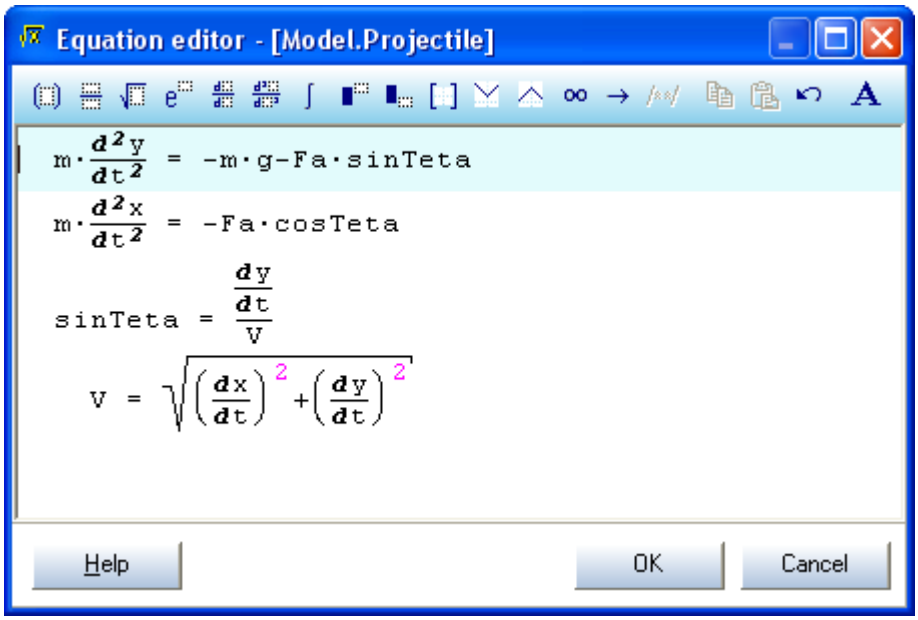


Figure 140 The "Text Editor" window

The **Text Editor** window, see **Figure 140**, consists of the editing area and the toolbar. The editing area allows the user to edit text. The toolbar provides shortcuts to most of the functions of the **Text Editor** window. When the **Text Editor** window is used to edit equations or expressions, the editing area displays user input in natural mathematical form.

3.7.1 Toolbar Buttons

The **Text Editor** window toolbar contains the following buttons:

– inserts parentheses.

– inserts a division.

– inserts a call to the `sqrt()` function.

– inserts a call to the `exp()` function.

– inserts a first derivative.

– inserts a second derivative.

– inserts a call to the `integral()` functor.

– inserts a superscript (power).

– inserts a subscript.

– inserts a fixed-size matrix.

– inserts a `minimize` operator.

– inserts a `maximize` operator.

– inserts the infinity symbol.

– inserts a right arrow.

– inserts comments.

– reverts the last action.

– opens the **Font** dialog box to edit the font used by the **Text Editor** window.

– allows the user to print the contents of the **Text Editor** window.

– allows the user to save the contents of the **Text Editor** window to a file as an image.

3.7.2 Editing Area

When the **Text Editor** window is used to edit raw text, the editing area functions like a conventional text editor. When editing equations and expressions, the editing area displays user input in natural mathematical form. When editing functions and procedures, an option to use natural mathematical display is also available and can be turned on and off as necessary.

Some of the functions of the **Text Editor** window are available via the context menu, which can be opened by right-clicking anywhere in the editing area. The context menu contains the following items:

Mathematical style – toggles natural mathematical display. This item is only available when editing a function or procedure.

Word-wrap – toggles word wrapping.

Syntax highlighting – toggles syntax highlighting.

Font… – opens the **Font** dialog box to edit the font used by the **Text Editor** window.

Print… – allows the user to print the contents of the **Text Editor** window.

Save image to file... – allows the user to save the contents of the **Text Editor** window to a file as an image.

Find… – opens the **Find** dialog box to search for text.

Find next – finds the next occurrence of the string entered in the **Find** dialog box.

3.8 COMPONENT EDITOR

The **Component Editor** window allows the user to edit record or connector component values.

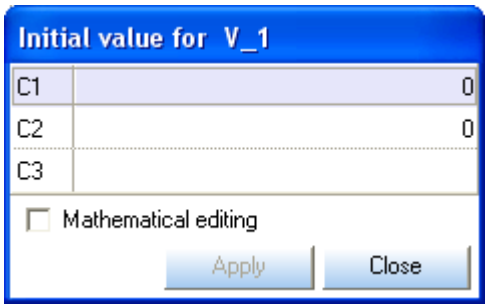


Figure 141 The "Component Editor" window

The **Component Editor** window looks as shown in **Figure 141**. The window displays components in a table, one row per component. Component names are displayed in the first column. Their respective values are displayed in the second column.

Every value can be edited either in-place or using the **Text Editor** window (see **Section 3.7**). This is controlled by the **Mathematical editing** checkbox.

Modified values are labeled with the sign, as shown in **Figure 142**.

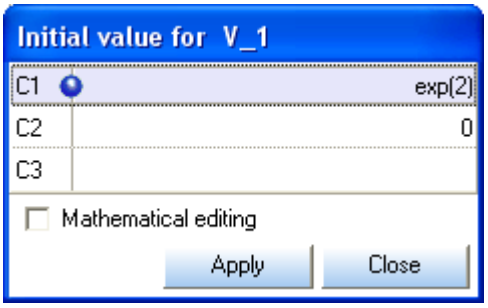


Figure 142 The "Component Editor" window showing a modified value

The **Apply** button applies the current changes and closes the window. The **Close** button closes the window without applying the current changes.

❖ **To edit a value in-place:**

1. Make sure **Mathematical editing** is unchecked.
2. Double-click the row you want to modify.
   An in-pace editing field will appear.
3. Edit the value.
4. Hit *Enter* or click anywhere outside the editing field.

❖ **To edit a value in the "Text Editor" window:**

1. Make sure **Mathematical editing** is checked.
2. Double-click the row you want to modify.
   The **Text Editor** window will appear.
3. Use the **Text Editor** window to edit the value.

❖ **To apply the current changes and close the "Component Editor" window:**

1. Click **Apply**.

❖ **To close the "Component Editor" window without applying the current changes:**

1. Click **Close**.

3.9 CALCULATOR

The **Calculator** window allows the user to evaluate expressions involving parameters (see **Section 2.8**), variables (see **Section 2.9**), constants (see **Section 2.10**), and predefined functions (see **Section 2.14.1**). The **Calculator** window looks as shown in **Figure 143**.

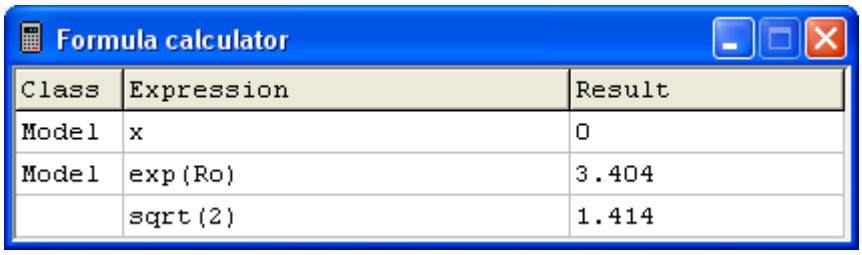


Figure 143 The "Calculator" window

❖ **To open the "Calculator" window:**

1. Click **Calculator** on the main toolbar, or
   Choose **Tools | Calculator** from the main menu.

❖ **To enter an expression for evaluation:**

1. Select a row in the **Calculator** window.
2. If the expression will involve parameters, variables, or constants of some class, specify the class. To do that, right-click the selected row and choose **Choose class...** from the popup menu.
   The **Choose class** dialog box will appear, as shown in **Figure 144**.
   In the **Choose class** dialog box, select the class and click **OK**.
   The name of the selected class will appear in the first cell of the row.
   Note, you can choose at most one class per row.

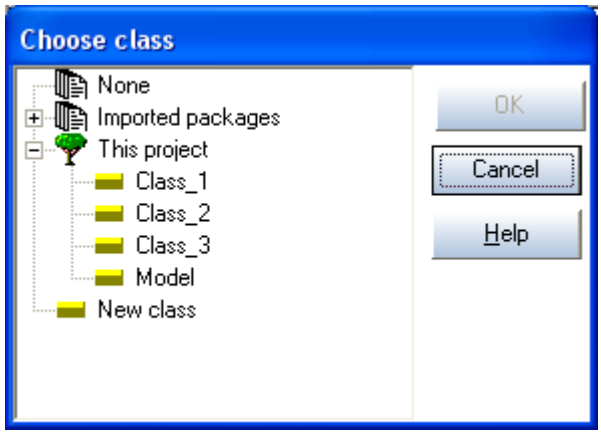


Figure 144 The "Choose class" dialog box

3. Click the second cell of the row and type the expression you want to evaluate. In the expression, you can use parameters, variables, and constants of the class specified on the previous step as well as any predefined functions. If you want to find out the value of a parameter, variable, or constant, simply type the name of the parameter, variable, or constant.
4. Hit *Enter*.
   The result of the expression will appear in the third cell as shown in **Figure 145**.

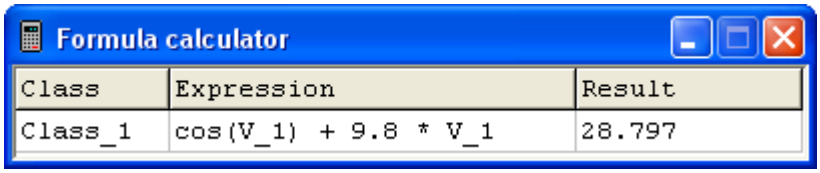


Figure 145 Evaluation result

❖ **To evaluate an already entered expression:**

1. Right-click the corresponding row and choose **Calculate row** from the popup menu.

❖ **To evaluate all available expressions:**

1. Right-click anywhere in the **Calculator** window and choose **Calculate all** from the popup menu.

❖ **To add a row:**

1. Right-click anywhere in the **Calculator** window and choose **Add line** from the popup menu.

❖ **To remove a row:**

1. Right-click the row and choose **Delete line** from the popup menu.

3.10 TEXTUAL PROJECT REPRESENTATION

In Rand Model Designer, every project has a textual representation in a language called MVL, see **Section 6.3**. At any time, the textual representation of the currently open project can be saved to or loaded from a text file. This capability can be used for various purposes including advanced editing and integration with other software.

By convention, MVL files are given the extension `.mvl`.

❖ **To save the textual representation of the currently open project into a file:**

1. Choose **Project | Save as text...** from the main menu.
   The **Export project as text to** dialog box will appear as shown in **Figure 146**.

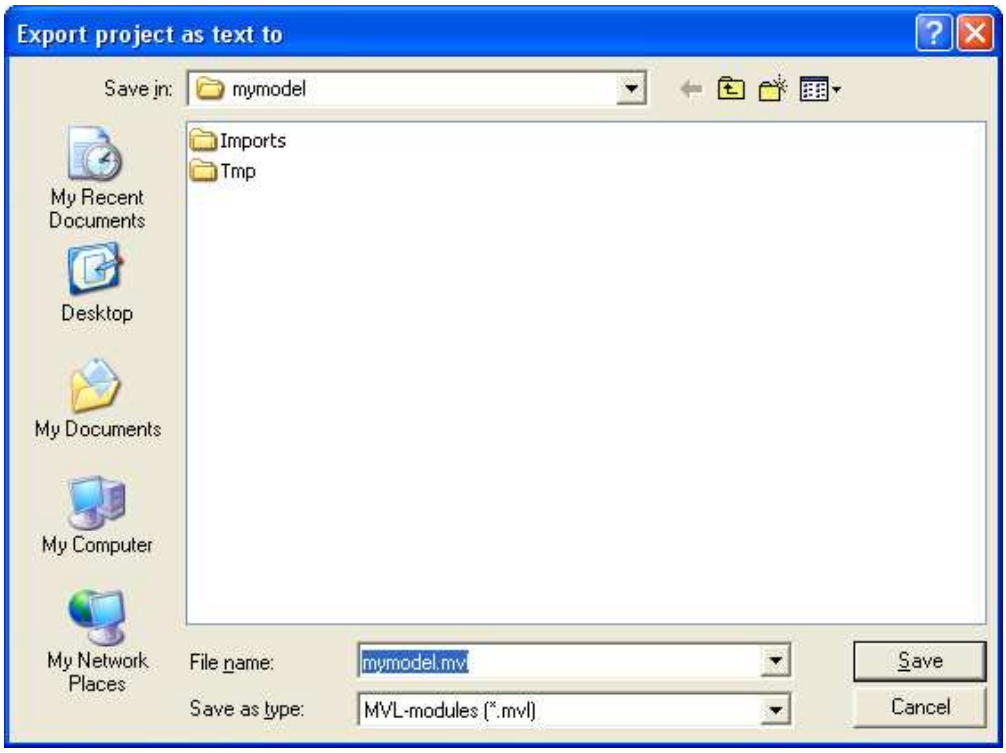


Figure 146 The "Export project as text to" dialog box

2. Browse to the folder where you want to save the textual representation.

3. Type the name of the file in the **File name** field.

4. Click **Save**.
   The textual representation of the project will be saved to the specified file. The resulting file will be opened in the default text editor, as shown in **Figure 147**.

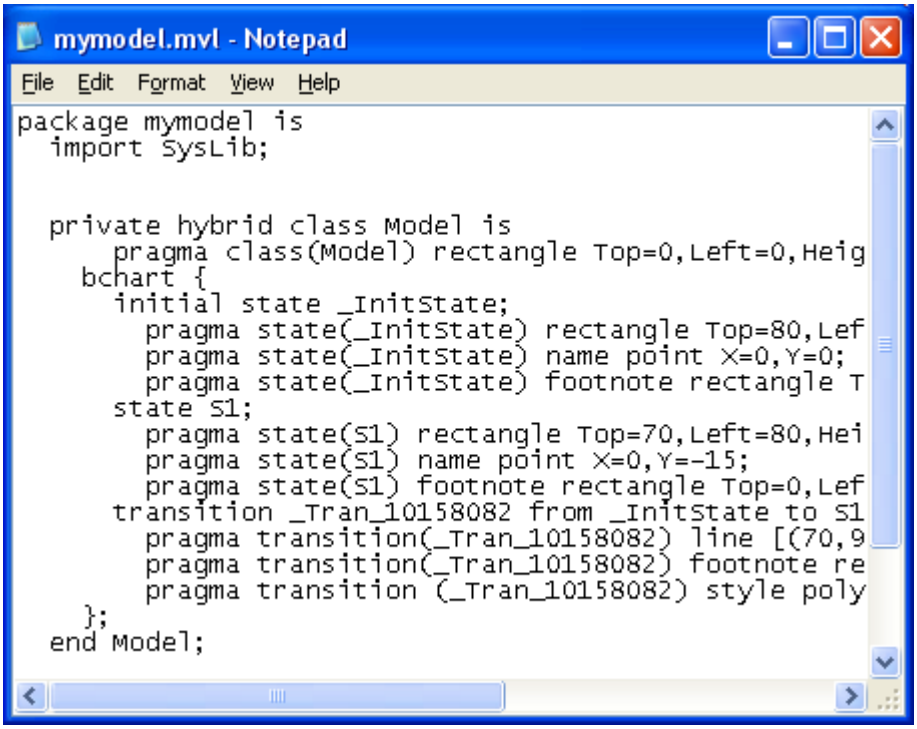


Figure 147 Textual representation of a project

❖ **To load a textual representation into the currently open project:**

1. Choose **Project | Load from text…** from the main menu.
   The **Import project as text from** dialog box will appear as shown in **Figure 148**.

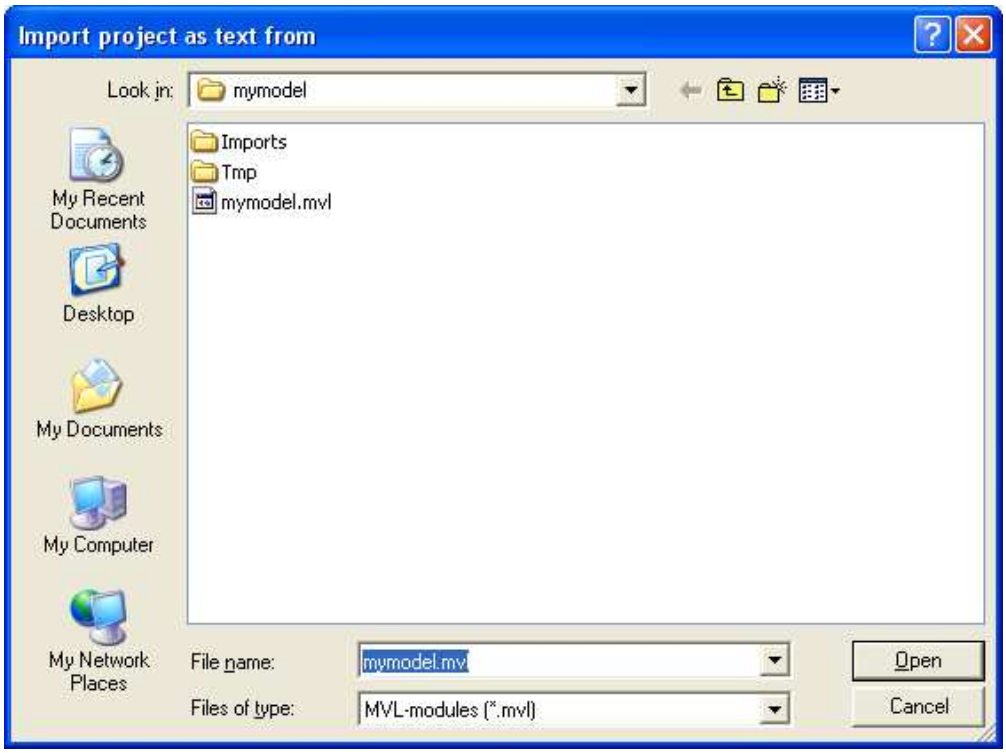


Figure 148 The "Import project as text from" dialog box

2. Select the file you want to load.
3. Click **Open**.
   Rand Model Designer will parse the selected file.
4. If there are no problems, the success message will appear as shown in **Figure 149**.
   Click **OK**.
   The Rand Model Designer will re-generate and re-open the project.

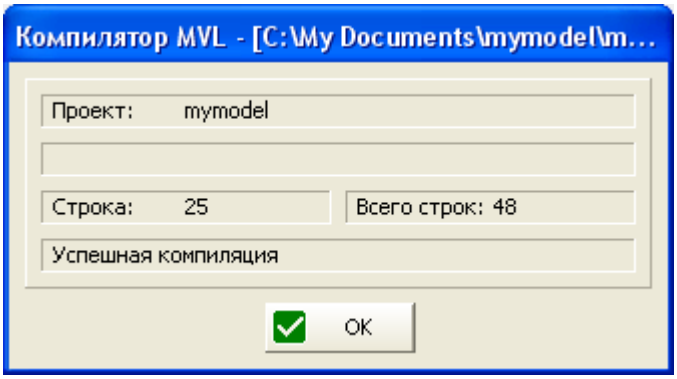


Figure 149 Success message

5. If there is a problem, a window describing the encountered error will appear as shown in **Figure 150**.
   In this window, you can either try to correct the problem and retry the operation by clicking **Retry**, or cancel the operation by clicking **Cancel**.

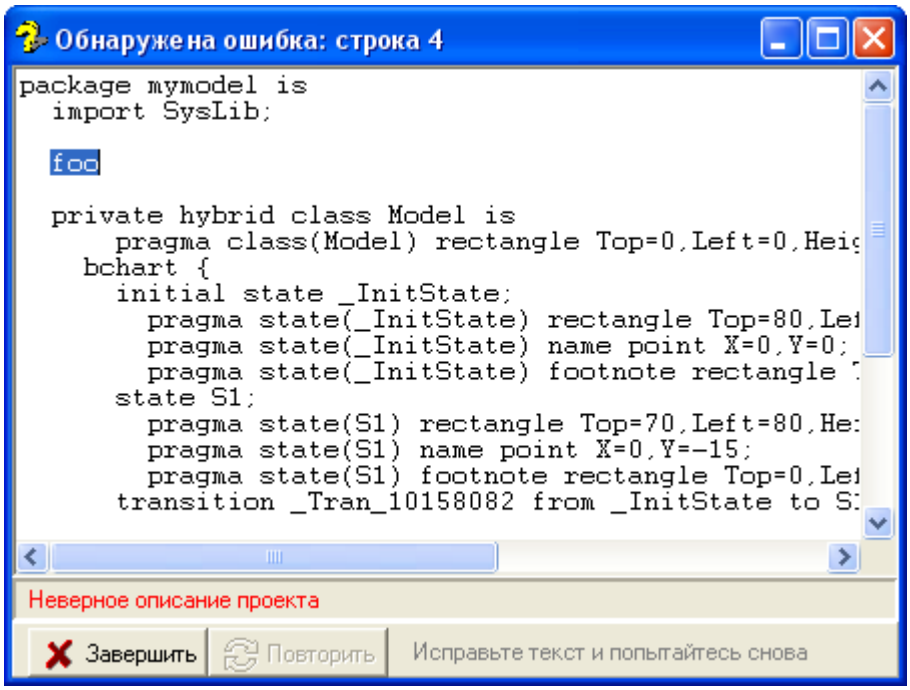


Figure 150 The "Error" window

3.11 WORKING WITH EXECUTABLE MODELS

Whenever there is a project open in the integrated modeling environment, the user can at any time run it by clicking the **Run model** toolbar button or choosing the respective item from the main menu: this automatically generates and starts the visual executable model. The visual executable model is a separate Windows application that implements the model and allows the user to experiment with it. The visual executable model can at any time be terminated from either the integrated modeling environment or the visual executable model itself. This functionality is available in all Rand Model Designer versions.

Additionally, Rand Model Designer Corporate and Rand Model Designer Educational versions allow the user to build and save the executable model as a standalone executable that can be run separately from the integrated modeling environment. Furthermore, the project can be configured to produce the embeddable executable model instead of the visual one, see **Section 3.12.3**. The embeddable executable model is a Windows DLL that any software can link to in order to get programmatic access to the model.

For more on executable models, see **Chapter 4**.

❖ **To run the visual executable model:**

1. Click **Run model** on the main toolbar, or
   Choose **Model | Run** from the main menu.
   The visual executable model will appear.

❖ **To terminate the visual executable model:**

1. Click **Stop model** one the main toolbar, or
   Choose **Model | Stop** from the main menu.
   The visual executable model will disappear.

❖ **To build the standalone executable model (available in Rand Model Designer Corporate and Rand Model Designer Educational):**

1. Click **Build model** on the main toolbar, or
   Choose **Model | Build** from the main menu.
   Rand Model Designer will build the standalone executable model and display a success message.
2. Click **OK**.

❖ **To save the standalone executable model with a specific name and location (available in Rand Model Designer Corporate and Rand Model Designer Educational):**

1. Choose **Model | Save as…** from the main menu.
   The **Save model as** dialog box will appear as shown in **Figure 151**.

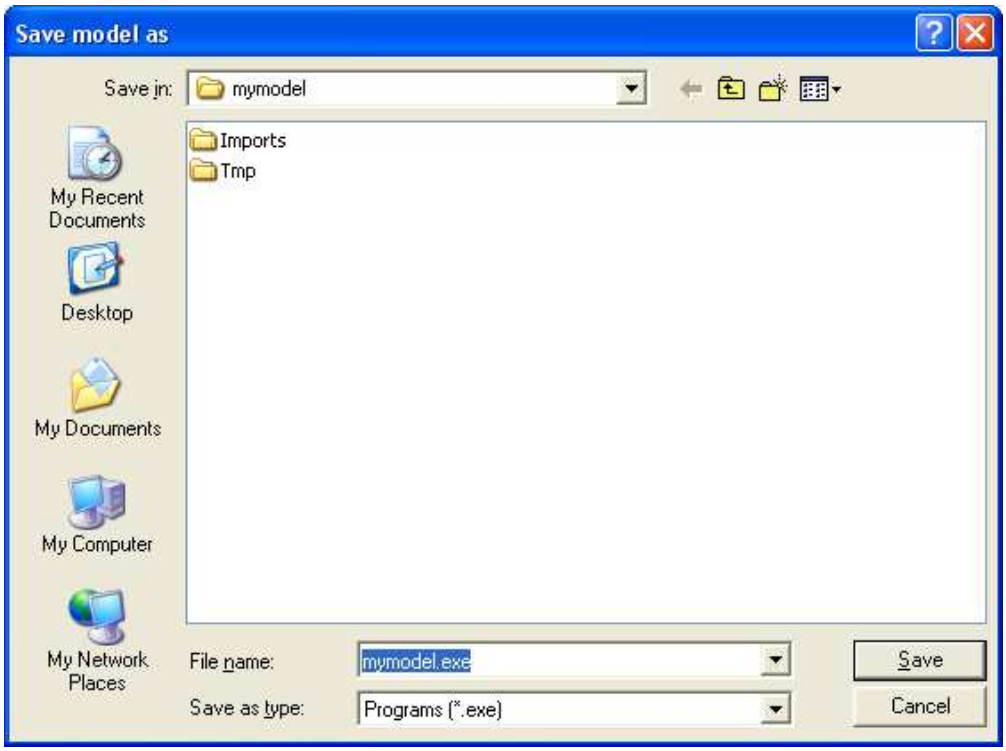


Figure 151 The "Save model as" dialog box

2. Browse to the folder where you want to save the executable model.
3. Type the name of the file in the **File name** field.
4. Click **Save**.

3.12 THE "PROJECT OPTIONS" DIALOG BOX

The **Project options** dialog box allows the user to edit properties of the currently open project.

❖ **To open the "Project options" dialog box:**

1. Click **Project options** on the main toolbar, or
   Choose **Project | Options…** from the main menu.
   The **Project options** dialog box will appear as shown in **Figure 152**.

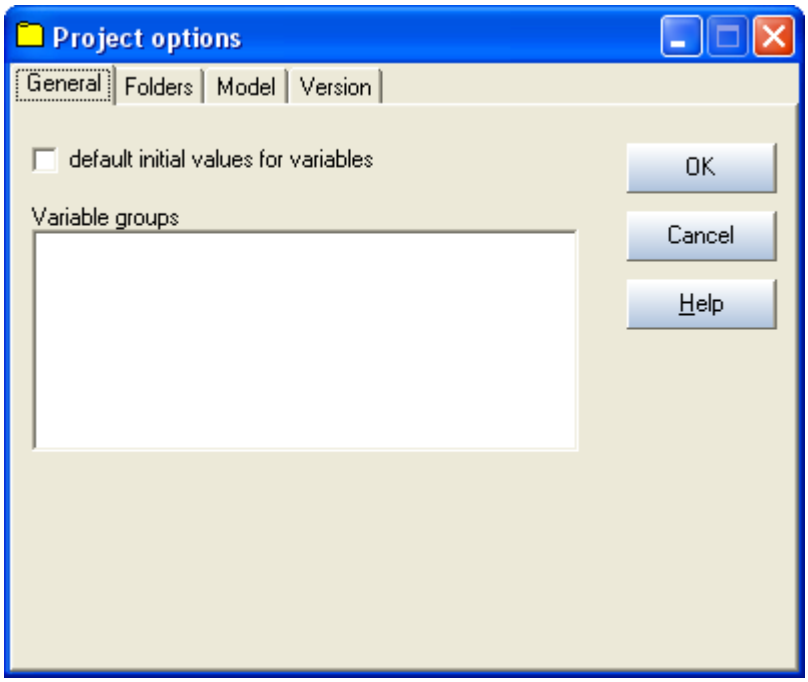


Figure 152 The “Project options” dialog box, the “General” tab

The **Project options** dialog box consist of the following tabs: **General**, **Folders**, **Model**, and **Version**. The subsections that follow describe the contents of these tabs.

3.12.1 The “General” Tab

This tab contains the following elements:

Default initial values for variables – If this checkbox is checked, Rand Model Designer automatically generates a reasonable default value for each new parameter, variable, or constant created by the user. Otherwise, the value of each new parameter, variable, or constant remains undefined unless explicitly specified by the user.

Variable groups – This is a list of variable groups defined in the project, see **Section 3.6**. The context menu, which can be opened by right-clicking in the list, provides the **Add group** and **Delete group** items that allows the user to add and remove groups. This function is only available in Rand Model Designer Corporate and Rand Model Designer Educational.

3.12.2 The “Folders” Tab

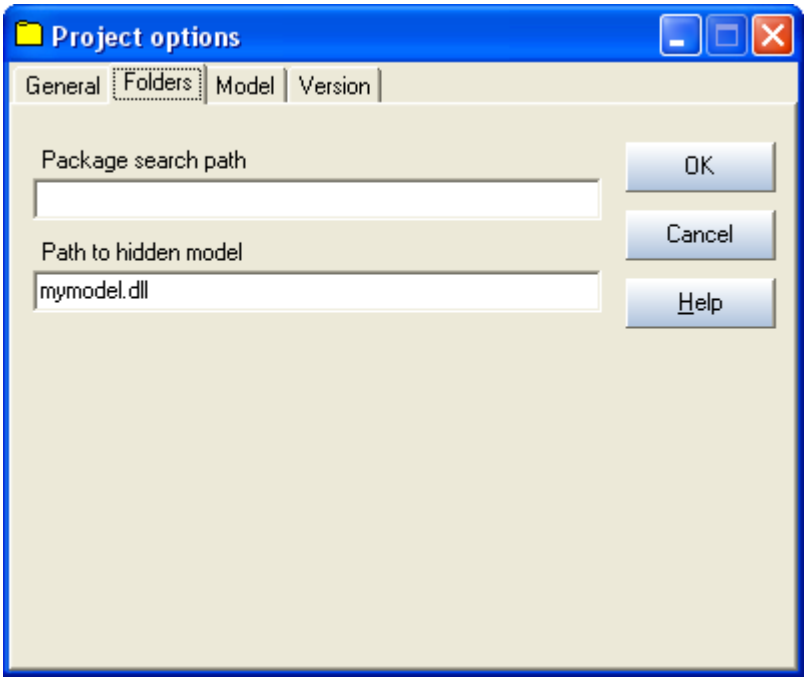


Figure 153 The “Folders” tab

This tab contains the following elements:

Package search path – This field specifies the set of folders where Rand Model Designer looks for imported packages. The field is populated automatically as imported packages are added to the project (see **Section 3.3.6**). However, if an already imported package moves to a new location, this field needs to be modified manually to let the importing project know the new location.

Path to hidden model – This field specifies the path used to store the generated embeddable model, see **Section 4.3**. The path is relative to the project folder.

3.12.3 The “Model” Tab

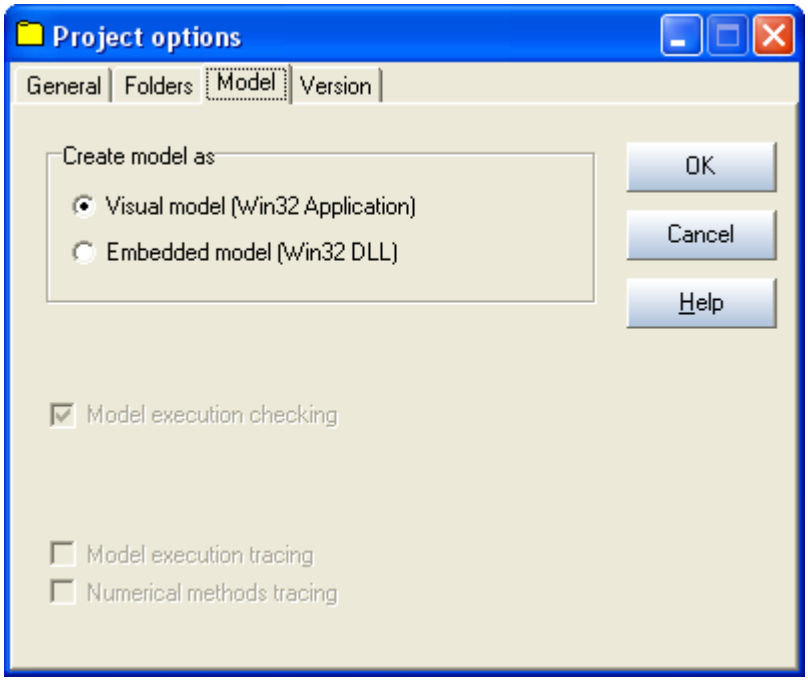


Figure 154 The “Model” tab

This tab contains the following elements:

Visual model (Win32 Application) – This option tells Rand Model Designer to generate the visual executable model when building the project, see **Chapter 4**. This is the default setting for new projects.

Embedded model (Win32 DLL) – This option tells Rand Model Designer to generate the embeddable executable model when building the project, see **Chapter 4**. This option is only available in Rand Model Designer Corporate and Rand Model Designer Educational.

Model execution checking – This checkbox enables/disables embeddable model execution checking.

Model execution tracing – This checkbox enables/disables tracing in the embeddable model.

Numerical methods tracing – This checkbox enables/disables tracing of numerical methods in the embeddable model.

3.12.4 The “Version” Tab

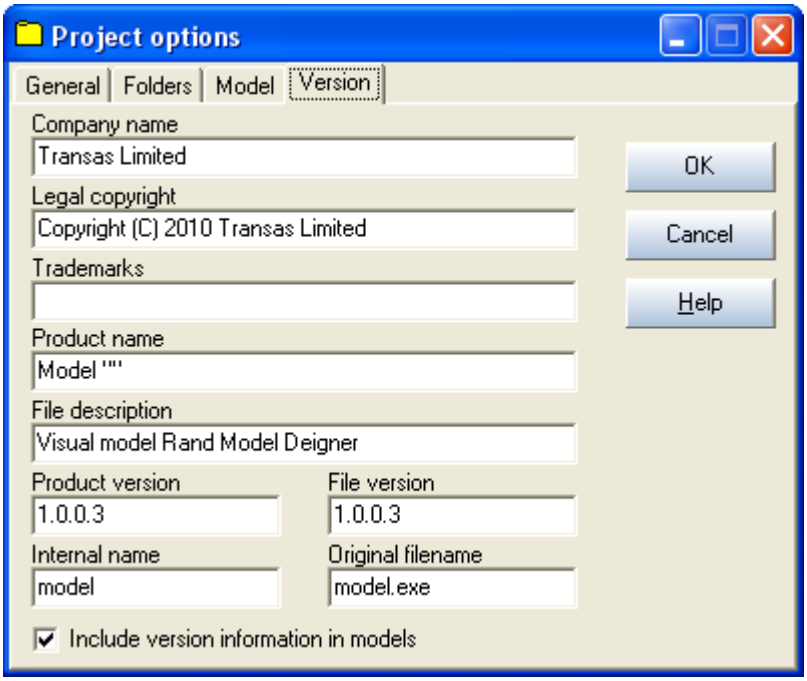


Figure 155 The “Version” tab

This tab contains the following elements:

Company name – This field specifies the name of the company that produced the model.

Legal copyright – This field specifies the copyright notices that apply to the model.

Trademarks – This field specifies the trademarks and registered trademarks that apply to the model.

Product name – This field specifies the name of the product with which the model is distributed.

File description – This field contains a brief description of the model.

Product version – This field specifies the version of the product with which the model is distributed.

File version – This field specifies the version of the model.

Internal name – This field specifies the internal name of the model.

Original filename – This field specifies the original name of the model.

Include version information in models – If this checkbox is checked, Rand Model Designer includes the version information specified on this tab in the executable model generated from this project. Otherwise, the version information is not included in the executable model.

3.13 THE “ENVIRONMENT OPTIONS” DIALOG BOX

The **Environment options** dialog box allows the user to customize the integrated modeling environment.

❖ **To open the "Environment options" dialog box:**

1. Choose **Tools | Environment options…** from the main menu.
   The **Environment options** dialog box will appear as shown in **Figure 156**.

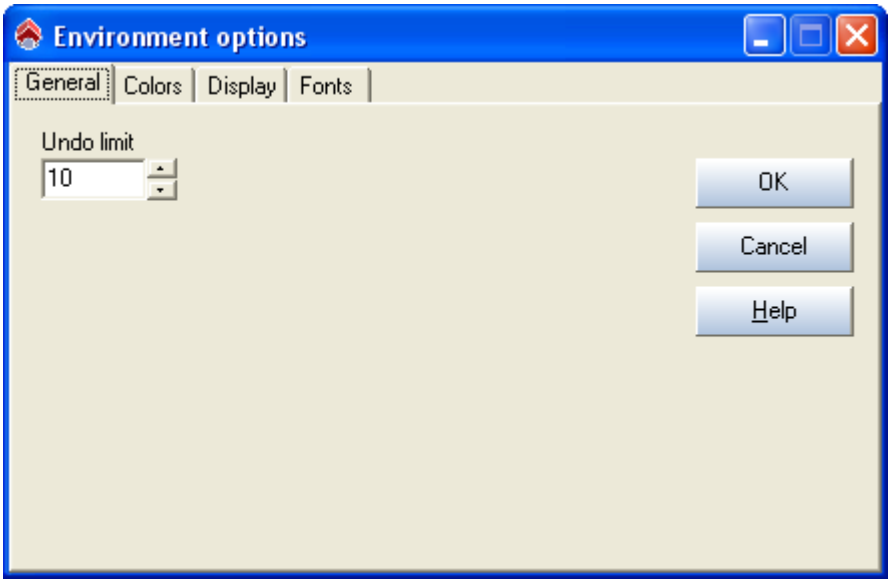


Figure 156 The "Environment options" dialog box, the "General" tab

The **Environment options** dialog box consists of the following tabs: **General**, **Colors**, **Display**, and **Fonts**. The subsections that follow describe the contents of these tabs.

3.13.1 The "General" Tab

This tab contains the following element:

Undo limit – This field specifies the maximum undo buffer size, that is, the maximum number of actions that can be undone using the **Edit | Undo** command.

3.13.2 The "Colors" Tab

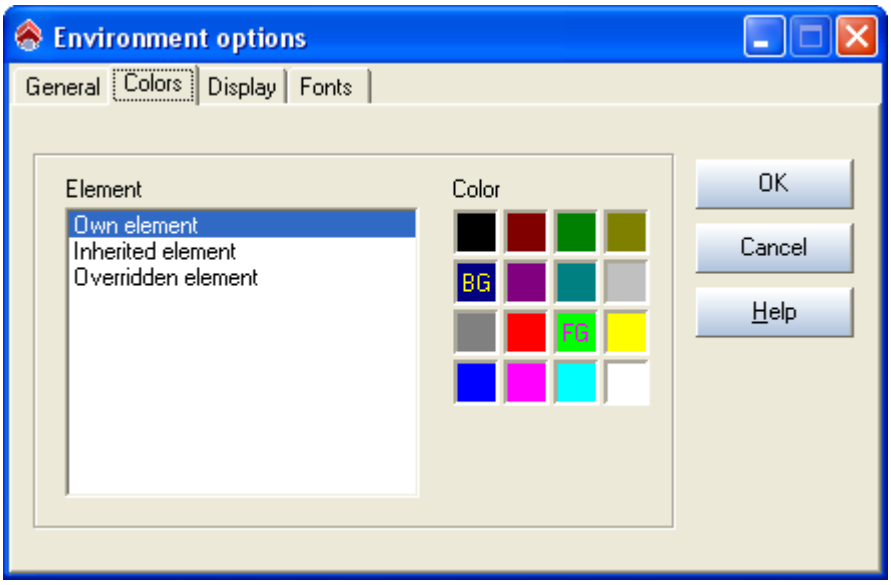


Figure 157 The "Colors" tab

The **Colors** tab, see **Figure 157**, allows the user to configure colors for different categories of project elements.

Element categories appear in the **Element** list. The available categories are: **Own element**, **Inherited element**, and **Overridden element**. The **Own element** category represents new elements of a class or project. The **Inherited element** and **Overridden element** categories represent inherited and overridden elements respectively. To view or modify colors for a particular category, select that category in the **Element** list.

For every category, two colors can be selected in the **Color** palette: foreground and background. The currently selected foreground and background colors are denoted by the **FG** and **BG** labels, respectively. To select a color as a foreground color, click that color in the **Color** palette. To select a color as a background color, right-click that color in the **Color** palette.

3.13.3 The “Display” Tab

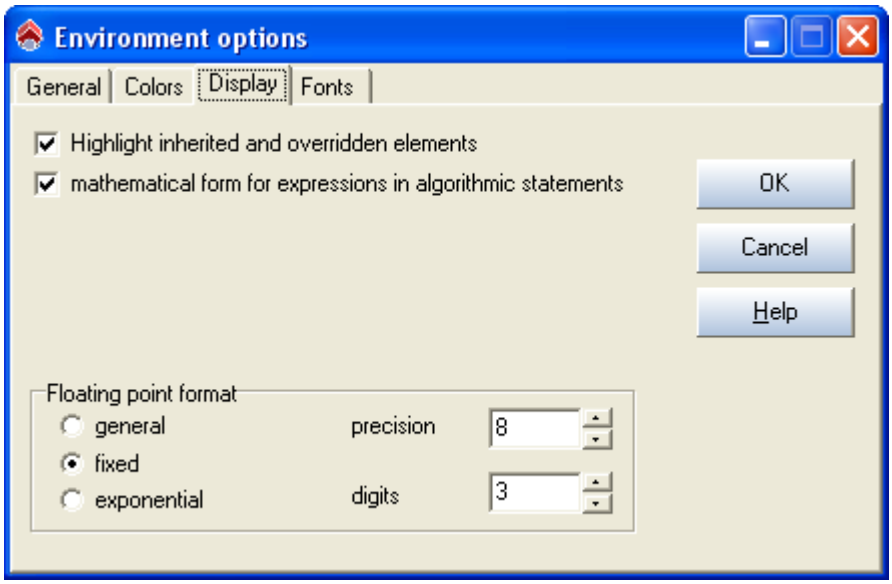


Figure 158 The “Display” tab

This tab contains the following elements:

Highlight inherited and overridden elements – This checkbox enables/disables highlighting of inherited and overridden elements on behavior charts and structure diagrams.

Mathematical form for expressions in algorithmic statements – This checkbox determines whether the **Mathematical style** option of the **Text Editor** window is on or off by default, see **Section 3.7.2**.

Floating point format – This group of options and fields defines the format used to display floating-point numbers everywhere in the integrated modeling environment. The **general**, **fixed**, and **exponential** options select the general, fixed, and exponential floating-point formats respectively. The **precision** field specifies the maximum number of digits in the textual representation of floating-point numbers. The **digits** field specifies the number of decimal places.

3.13.4 The “Fonts” Tab

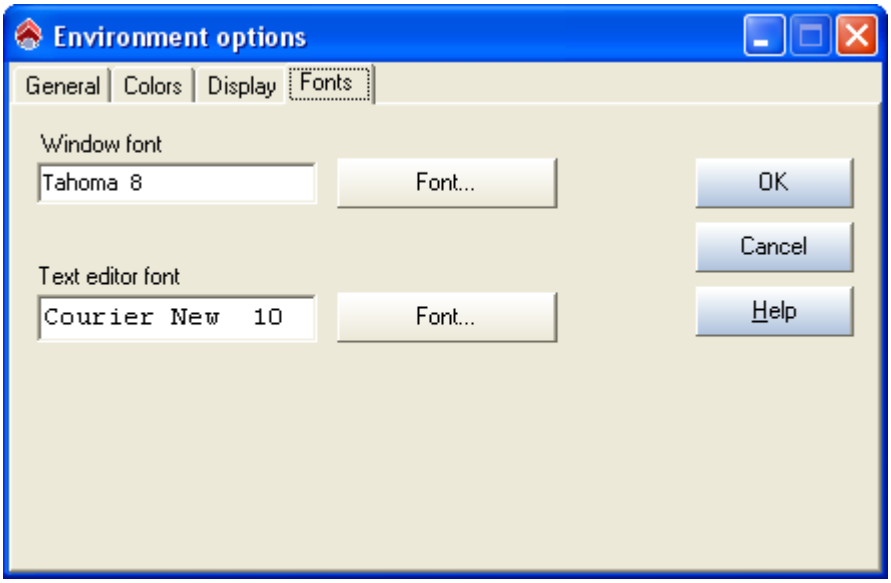


Figure 159 The “Fonts” tab

This tab contains the following elements:

Window font – This field specifies the font used by all windows of the integrated modeling environment except for the **Text Editor** window. To modify the font, click the **Font** button and use the standard **Font** dialog box to choose the font.

Text editor font – This field specifies the font used by the **Text Editor** window. To modify the font, click the **Font** button and use the standard **Font** dialog box to choose the font.

3.14 MISCELLANEOUS FUNCTIONS

Like many applications, the integrated modeling environment allows the user to:

Undo recent actions

Cascade, tile, and minimize open windows

Print the contents of windows

Consult the built-in help

View application information

Quit the application

❖ **To undo the last action:**

1. Click **Undo** on the main toolbar, or
   Choose **Edit | Undo** from the main menu, or
   If you are currently in the **Text Editor** window, click **Undo** on the **Text Editor** window toolbar.

❖ **To cascade open windows:**

1. Choose **Window | Cascade** from the main menu.

❖ **To tile open windows:**

1. Choose **Window | Tile** from the main menu.

❖ **To minimize open windows:**

1. Choose **Window | Minimize all** from the main menu.

❖ **To print the contents of the active window:**

1. Choose **Window | Print...** from the main menu.
   The standard **Print** dialog box will appear.
2. Adjust printing properties, if necessary, and click **OK**.

❖ **To open the help:**

1. Choose **Help | Rand Model Designer Help** from the main menu.

❖ **To view information about the application:**

1. Choose **Help | About...** from the main menu.

❖ **To quit the integrated modeling environment:**

1. Choose **Project | Exit** from the main menu.
2. If the currently open project contains unsaved changes, a confirmation message box will appear.
   If you want to save the changes, click **Yes**. Otherwise, click **No**.

CHAPTER 4
Executable Model

This chapter describes the categories and capabilities of executable models

An *executable model* is a program that is automatically created by Rand Model Designer from a project. Every executable model consists of the Rand Model Designer runtime and an instance of the `Model` class defined in the project.

4.1 EXECUTABLE MODEL CATEGORIES

Rand Model Designer can create executable models of two types: *visual* and *embeddable*.

The visual executable model is a Windows GUI application that can be run either from the integrated modeling environment or standalone. The visual executable model is usually used to debug and demonstrate model behavior. In addition to the minimal Rand Model Designer runtime, the visual executable model includes the GUI front-end that allows the user to:

Create time and phase diagrams

Create interactive 2D and 3D animations

Animate behavior charts

Debug the model using breakpoints

The visual executable model is discussed in **Section 4.2**.

The embeddable executable model is a Windows DLL that does not include any visualization capabilities. The embeddable executable model is intended for inclusion in other applications that can support their own visualization.

The embeddable executable model is discussed in **Section 4.3**.

4.2 VISUAL EXECUTABLE MODEL

When the user starts the visual executable model (see **Section 3.11**), the model's main window appears as shown in **Figure 160**.

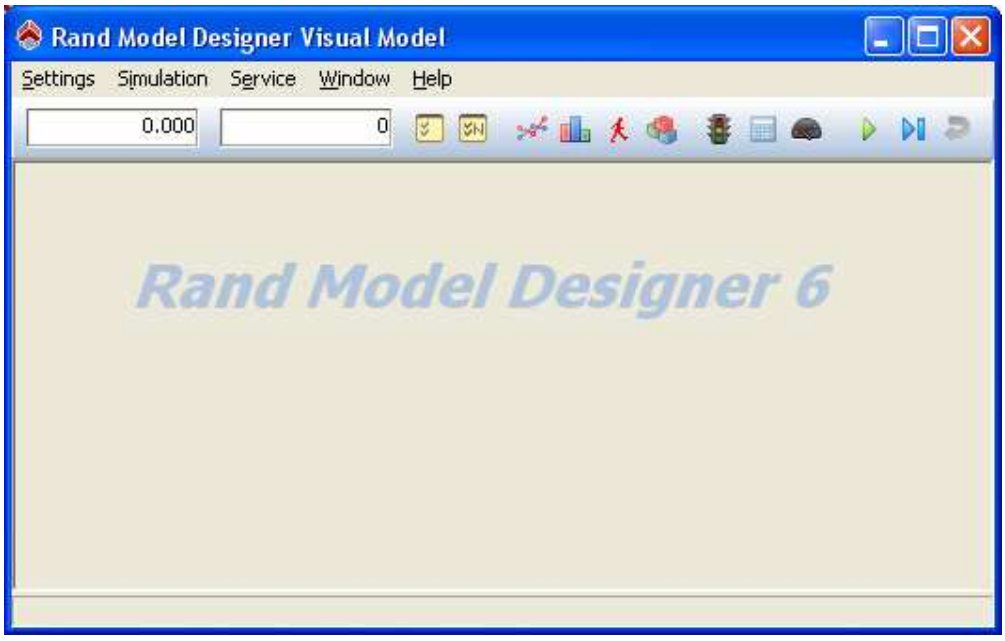


Figure 160 Visual executable model main window

The main window consists of the main menu, main toolbar, and working area. The main menu provides access to most of the functions of the model. The main toolbar

displays the current continuous and discrete model time and provides shortcuts to some of the most commonly used items of the main menu. The working area is where all other windows of the visual executable model are opened.

Depending on the model, the working area of a newly started model may initially contain some of the following windows:

A **Variables** window displaying all parameters and variables of the model object

A **Diagram** window displaying the time diagram of all variables of the model object

A **Behavior Chart** window displaying the behavior chart of the model object

A **Structure** window displaying the structure diagram of the model object

In addition to these default windows, the user can manually open more **Variables**, **Diagram**, **Histogram**, **Behavior Chart**, **Structure**, **2D Animation**, and **3D Animation** windows.

4.2.1 The "Variables" Window

The **Variables** window displays the current values of parameters and variables of some object (see **Section 2.8** and **Section 2.9**). The **Variables** window looks as shown in **Figure 161**.

Figure 161 The "Variables" window

The **Variables** window displaying the parameters and variables of the model object opens automatically upon executable model startup if the model object has any parameters or variables. The **Variables** windows of other objects can be opened manually from **Behavior Chart** (see **Section 4.2.2**) and **Structure** (see **Section 4.2.3**) windows.

The **Variables** window displays the values in a list, one row per parameter or variable. Parameter and variable names are shown on the left and their respective values are shown on the right.

Vector and matrix values are displayed as vector and matrix literals. If a vector or matrix value does not fit in the available space, the **Variables** window displays it partially and allows the user to view the full contents in a separate dialog box.

❖ **To view the full contents of a vector or matrix:**

1. Double-click the corresponding row in the **Variables** window, or
   Right-click the corresponding row in the **Variables** window and choose **Edit...** from the popup menu.
   The **Elements** dialog box will appear as shown in **Figure 162**.

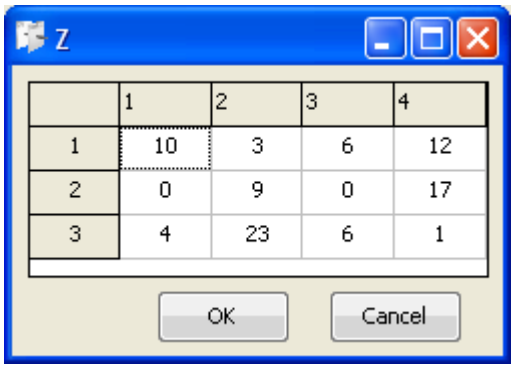


Figure 162 The "Elements" dialog box

Record and connector values (see **Section 2.13.3** and **Section 2.13.4**) are displayed as expandable nodes, as shown in **Figure 163**.

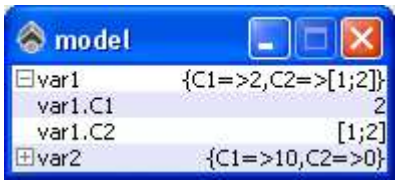


Figure 163 Record and connector values

The user can view the components of a record or connector by expanding the corresponding node or by opening the **Components** window.

❖ **To open the "Components" window for a record or connector:**

1. Double click the corresponding row in the **Variables** window, or
   Right-click the corresponding row in the **Variables** window and choose **Edit…** from the popup menu.
   The **Components** window will appear as shown in **Figure 164**.

Figure 164 The "Components" window

The **Variables** window allows the user to modify any scalar, vector, or matrix value by editing it in-place. In addition to this, vectors and matrices can by modified using the **Elements** dialog box, whereas records and connectors can be modified using the **Components** window. Variables can be modified at any time even while the model is running, whereas parameters can only be modified when the model is stopped (see **Section 4.2.14**).

❖ **To modify a scalar value in-place:**

1. Double-click the corresponding row in the **Variables** window, or
   Right-click the corresponding row in the **Variables** window and choose **Edit…** from the popup menu.
   An in-place editing field will appear as shown in **Figure 165**.

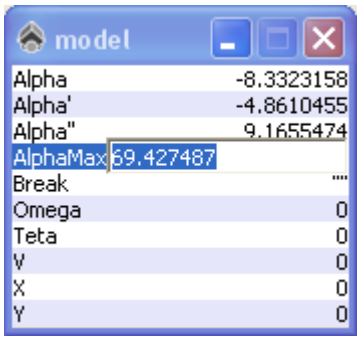


Figure 165 Editing the value of a variable

2. Type the new value.
3. Hit *Enter* or click anywhere outside the editing field.

❖ **To modify a vector or matrix in-place:**

1. Right-click the corresponding row in the **Variables** window and choose **Edit as text…** from the popup menu.
   An in-place editing field will appear.
2. Type the new value.
3. Hit *Enter* or click anywhere outside the editing field.

❖ **To modify a vector or matrix using the "Elements" dialog box:**

1. Double-click the corresponding row in the **Variables** window, or
   Right-click the corresponding row in the **Variables** window and choose **Edit…** from the popup menu.
   The **Elements** dialog box will appear as shown in **Figure 162**.
2. In the **Elements** dialog box, double-click the element you want to modify.
   An in-place editing field will appear.
3. Type a new value for the element.
4. Hit *Enter* or click anywhere outside the editing field.
5. If necessary, repeat the above steps to modify other elements.
6. Click **OK** in the **Elements** dialog box.

❖ **To modify a record or connector:**

1. Double-click the corresponding row in the **Variables** window, or
   Right-click the corresponding row in the **Variables** window and choose **Edit…** from the popup menu.
   The **Components** window will appear as shown in **Figure 164**.
2. In the **Components** window, double-click the component you want to modify.
   An in-place editing field will appear.
3. Type a new value for the component.
4. Hit *Enter* or click anywhere outside the editing field.
5. If necessary, repeat the above steps to modify other components.

4.2.1.1 Virtual Variables

A virtual variable is a variable that is defined in the visual executable model rather than the project. Every virtual variable has an associated expression specifying its

value. During model execution, the value of each virtual variable is updated automatically whenever the value of the associated expression changes.

Virtual variables are created and destroyed using the **Variables** window. In the **Variables** window, available virtual variables are displayed in red.

❖ **To create a new virtual variable:**

1. Right-click anywhere in the **Variables** window and choose **Create virtual variable…** from the popup menu.
   The **Virtual variable** dialog box will appear as shown in **Figure 166**.

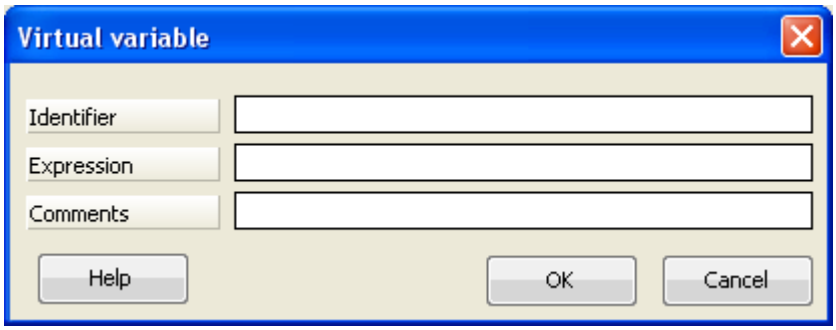


Figure 166 The "Virtual variable" dialog box

2. Type the name of the new virtual variable in the **Identifier** field.
3. Type the expression for the new virtual variable in the **Expression** field.
4. If necessary, type the comments in the **Comments** field.
5. Click **OK**.

❖ **To modify a virtual variable:**

1. Double-click the virtual variable, or
   Right-click the virtual variable and choose **Edit…** from the popup menu.
   The **Virtual variable** dialog box will appear as shown in **Figure 166**.
2. Modify the properties of the virtual variable as necessary.
3. Click **OK**.

❖ **To delete a virtual variable:**

1. Right-click the virtual variable and choose **Delete virtual variable…** from the popup menu.
   A confirmation message box will appear.
2. Click **Yes**.

4.2.2 The "Behavior Chart" Window

The **Behavior Chart** window visualizes the behavior chart of a hybrid object, see **Section 2.4**. The **Behavior Chart** window looks as shown in **Figure 167**.

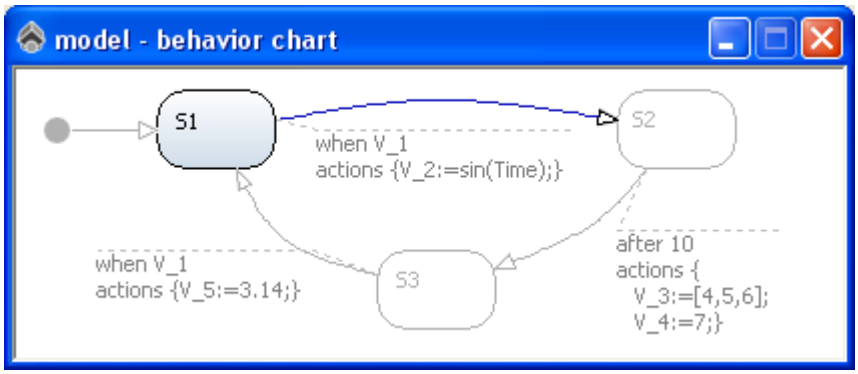


Figure 167 The "Behavior Chart" window

In the **Behavior Chart** window, all states and transitions are shown grayed out except for the current state and active transitions. When a transition is executed, it is highlighted orange. State and transition property labels can be turned on and off via the context menu.

❖ **To toggle property labels:**

1. Right-click anywhere in the **Behavior Chart** window and choose **Additional info** from the popup menu.

The **Behavior Chart** window of the model object opens automatically upon executable model startup if the model object is hybrid. The **Behavior Chart** windows of other hybrid objects can be opened manually as described below and in **Section 4.2.3**.

If a state displayed in a **Behavior Chart** window has a hybrid activity, the **Behavior Chart** window of the corresponding activity object can be opened as follows.

❖ **To open the "Behavior Chart" window of a hybrid state activity object:**

1. Double-click the state, or
   Right-click the state and choose **Behavior chart** from the popup menu.
   The **Behavior Chart** window of the activity object will appear as shown in **Figure 168**.

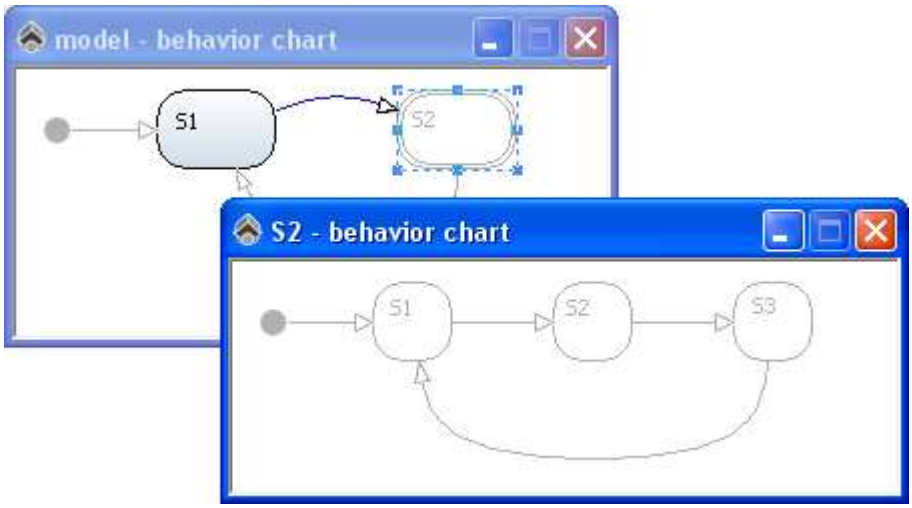


Figure 168 Opening the "Behavior Chart" window of a state activity object

If a state has an activity with parameters or variables, the **Variables** window of the corresponding activity object can be opened as follows.

❖ **To open the "Variables" window of a state activity object:**

1. Right-click the state and choose **Variables** from the popup menu.
   The **Variables** window of the activity object will appear as shown in **Figure 169**.

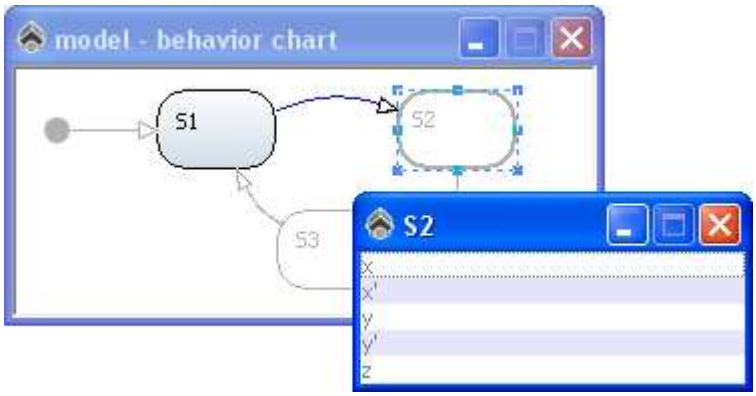


Figure 169 Opening the "Variables" window of a state activity object

If a state has a compound activity, the **Structure** window of the corresponding activity object can be opened as follows.

❖ **To open the "Structure" window of a compound state activity object:**

1. Right-click the state and choose **Local structure** from the popup menu.
   The **Structure** window of the activity object will appear as shown in **Figure 170**.

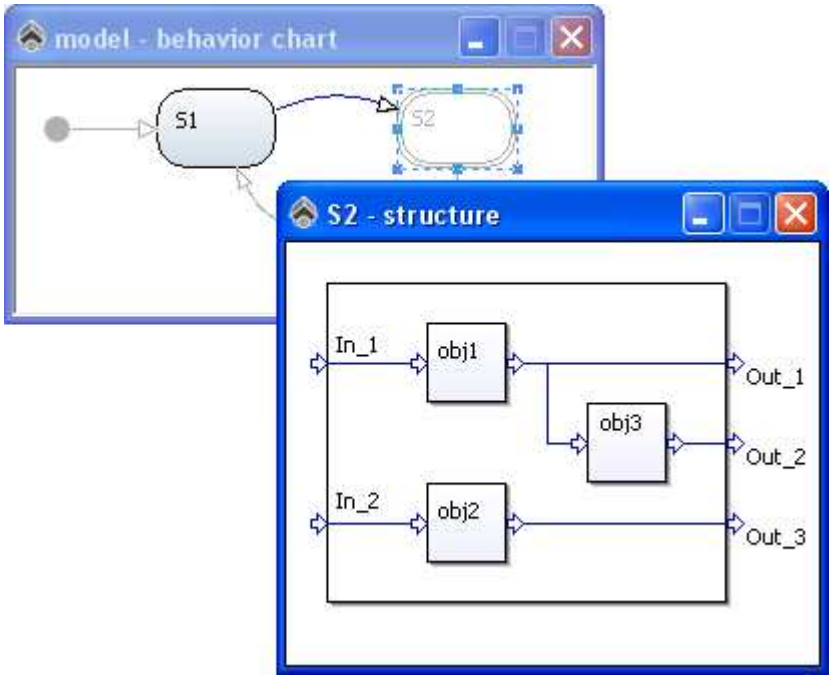


Figure 170 Opening the "Structure" window of a state activity object

If a state has internal transitions, the state's **Internal transitions** window can be opened as follows.

❖ **To open the "Internal transitions" window for a state:**

1. Right-click the state and choose **Internal transitions** from the popup menu.
   The **Internal transitions** window will appear as shown in **Figure 171**.

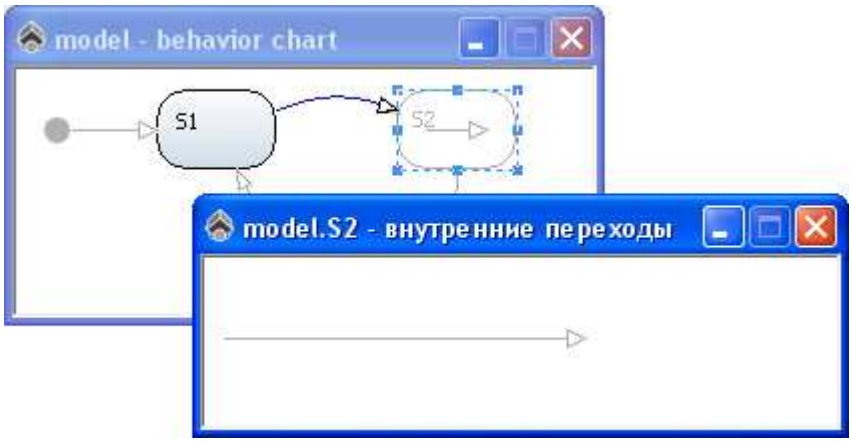


Figure 171 Opening the "Internal transitions" window of a state

The **Behavior Chart** window allows the user to associate a set of visual model settings with a state so that when the state becomes current, the corresponding model settings are automatically activated. This allows the user to configure the visual executable model to show different windows in different model states.

❖ **To associate a set of visual model settings with a state:**

1. Right-click the state and choose **Custom settings…** from the popup menu. The **Custom Settings** dialog box will appear as shown in **Figure 172**.

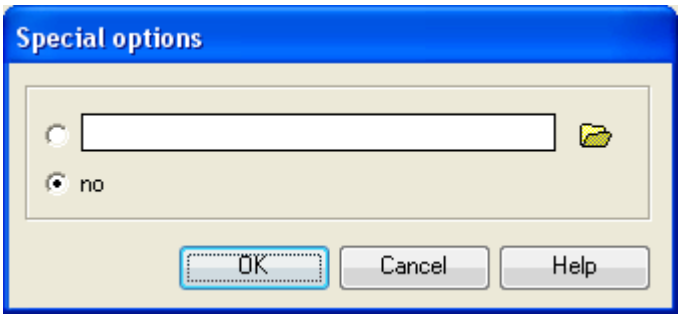


Figure 172 The "Custom settings" dialog box

2. Click .
   The **Open** dialog box will appear.
3. Select a model settings file and click **Open**.
   The selected name will appear in the **Custom Settings** dialog box.
4. Click **OK**.

❖ **To remove a set of visual model settings from a state:**

1. Right-click the state and choose **Custom settings…** from the popup menu. The **Custom Settings** dialog box will appear as shown in **Figure 172**.
2. Select the **no** option.
3. Click **OK**.

4.2.3 The "Structure" Window

The **Structure** window visualizes the structure diagram of a compound object, see **Section 2.7**. The **Structure** window looks as shown in **Figure 173**.

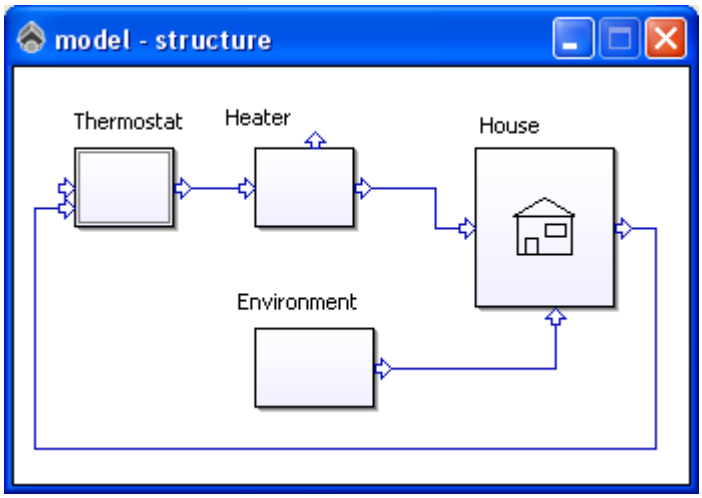


Figure 173 The "Structure" window

The **Structure** window is similar to the structure diagram editor (see **Section 3.4.4**) in that it displays all elements of the structure diagram of an object. However, the **Structure** window does not allow the user to modify the structure diagram.

In the **Structure** window, most elements have tooltips that display their properties. The tooltip of a local object displays the class name, actual parameter values, initial variable values, and comments associated with the object. The tooltip of an external variable displays the name of the variable and its current value. The tooltip of a link displays the names of the variables connected by the link.

If the **Show data exchange over links** option is set in the model settings (see **Section 4.2.13.1.2**), links are highlighted when the variables they connect change during model execution.

The **Structure** window of the model object opens automatically upon executable model startup if the model object is compound. The **Structure** windows of other compound objects can be opened manually as described below and in **Section 4.2.2**.

If a local object displayed in a **Structure** window is compound, its own **Structure** window can be opened as follows.

❖ **To open the "Structure" window of a compound local object:**

1. Double-click the local object, or
   Right-click the local object and choose **Local structure** from the popup menu.
   The object's **Structure** window will appear as shown in **Figure 174**.

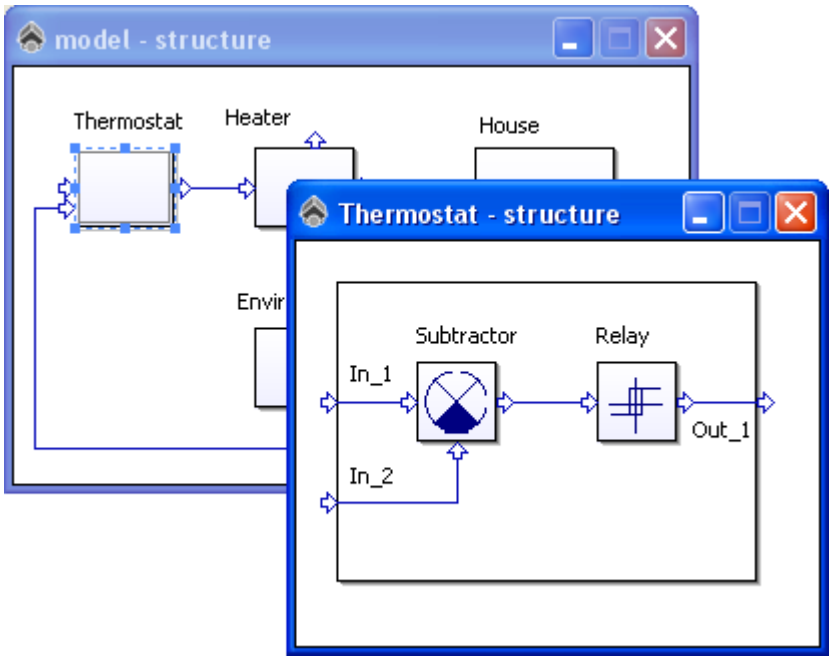


Figure 174 Opening the "Structure" window of a compound local object

If a local object is hybrid, its **Behavior Chart** window can be opened as follows.

❖ **To open the "Behavior Chart" window of a hybrid local object:**

1. Right-click the local object and choose **Behavior chart** from the popup menu. The object's **Behavior Chart** window will appear as shown in **Figure 175**.

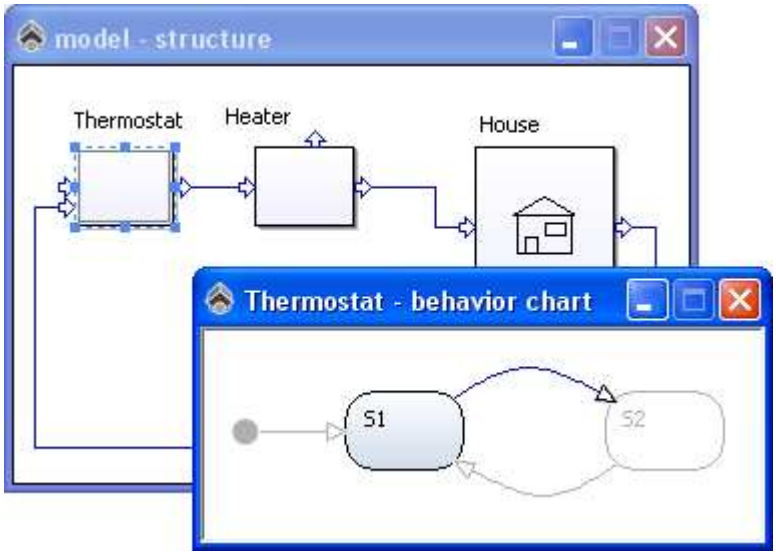


Figure 175 Opening the "Behavior Chart" window of hybrid local object

If a local object has parameters or variables, its **Variables** window can be opened as follows.

❖ **To open the "Variables" window of a local object:**

1. Right-click the local object and choose **Variables** from the popup menu. The object's **Variables** window will appear as shown in **Figure 176**.

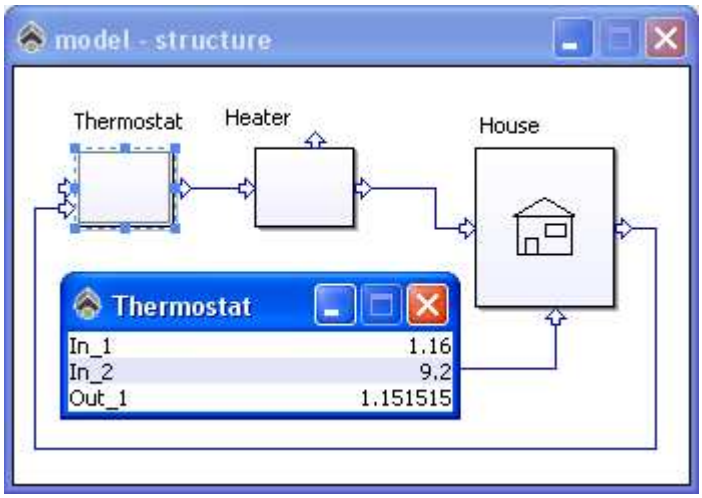


Figure 176 Opening the "Variables" window of a local object

The value of any external variable visible in a **Structure** window can be modified directly from the **Structure** window.

❖ **To modify the value of an external variable:**

1. Right-click the variable and choose **New value…** from the popup menu. The **New Value** dialog box will appear as shown in **Figure 177**.

Figure 177 The "New Value" dialog box

2. Type the new value for the variable.
3. Click **OK**.

The **Structure** window allows the user to add any of the standard 2D components (see **Section 4.2.6.1**) to the structure diagram and associate them with any of the available external variables. This way external variable values can be visualized in the **Structure** window.

❖ **To add a standard 2D component and associate it with an external variable:**

1. Open the **Standard 2D components** panel, see **Section 4.2.6.1**.
2. Drag a component of your choice from the **Standard 2D components** panel and drop it onto the variable in the **Structure** window. Alternatively, drag the component from the panel and drop it anywhere in the **Structure** window, then drag the variable and drop it onto the component.
   In either case, the component will appear in the **Structure** window as shown in **Figure 178** and will become associated with the corresponding variable.

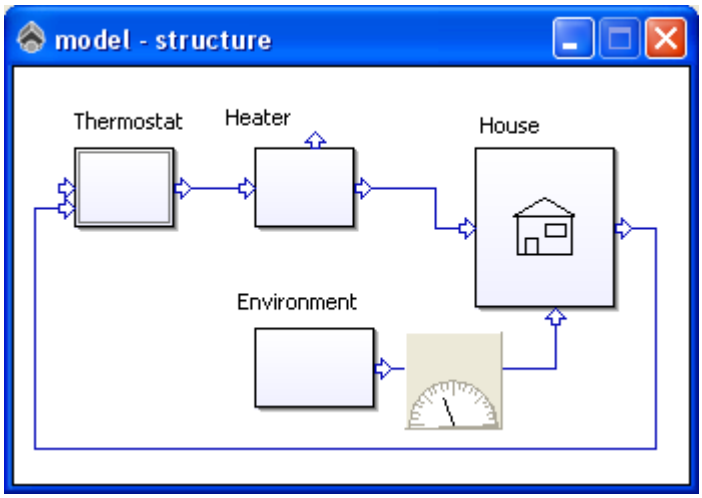


Figure 178 A "Structure" window with a standard 2D component

3. Move the created component to the desired location and, if necessary, configure it via its context menu. For more information, see **Section 4.2.6.1**.

4.2.4 The "Diagram" Window

The **Diagram** window displays the time or phase diagram for a set of variables.

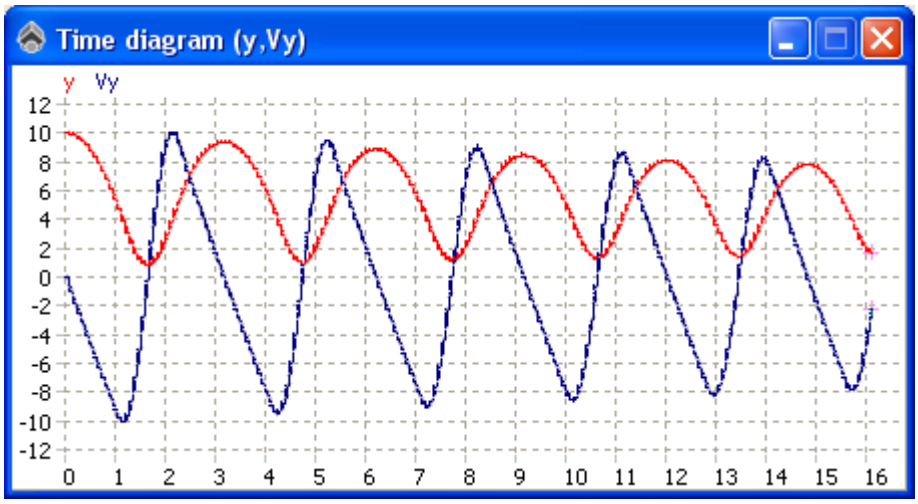


Figure 179 Typical time diagram

A typical time diagram is shown in **Figure 179**. In a time diagram, the x-axis represents model time while the y-axis represents variable values.

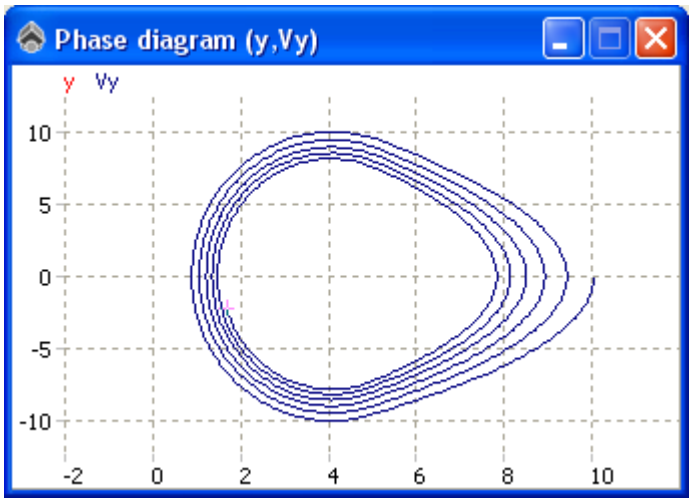


Figure 180 Typical phase diagram

A typical phase diagram is shown in **Figure 180**. In a phase diagram, the x-axis represents one variable while the y-axis represents the other variables.

A **Diagram** window displaying the time diagram of all variables of the model object opens automatically upon executable model startup if the model object has any displayable variables. Other **Diagram** windows can be opened and populated manually. The user can open as many **Diagram** windows as necessary.

❖ **To open a new "Diagram" window:**

1. Click **New diagram** on the main toolbar, or
   Choose **Window | New diagram** from the main menu.
   A new **Diagram** window will appear as shown in **Figure 181**.

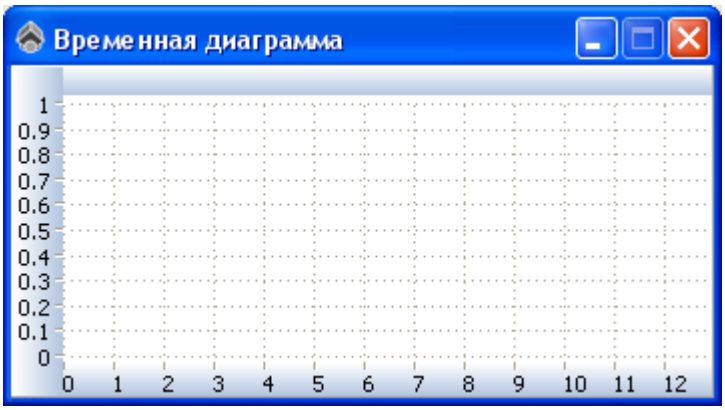


Figure 181 New "Diagram" window

The **Diagram** window can display `double`, `integer`, `short`, and `byte` variables, individual elements of vectors and matrices, enumeration variables, and signal variables.

The user populates the **Diagram** window with variables by dragging and dropping them from other windows.

❖ **To add a variable to a "Diagram" window:**

1. Drag the variable from a **Variables** or **Structure** window and drop it onto the **Diagram** window.
2. If the variable is a `double`, `integer`, `short`, `byte`, enumeration, or signal, it will be added to the diagram immediately.

Otherwise, if the variable is a vector, the **Select Vector Elements** dialog box will appear as shown in **Figure 182**. Or, if the variable is a matrix, the **Select Matrix Elements** dialog box will appear as shown in **Figure 183**.

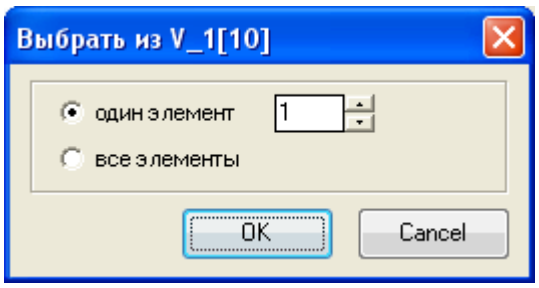


Figure 182 The "Select" dialog box for vectors

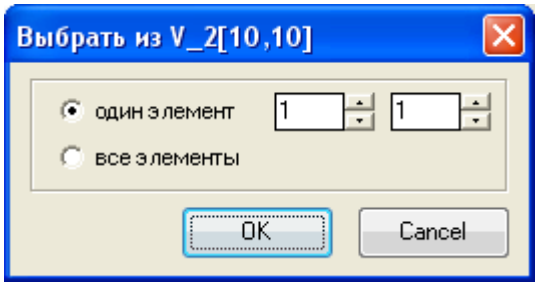


Figure 183 The "Select" dialog box for matrices

3. To add a single element of the vector or matrix, choose the **Single element** option and identify the element using the field(s) on the right.
4. To add all elements of the vector or matrix, choose the **All elements** option.
5. Click **OK**.
   The selected element(s) will be added to the **Diagram** window.

Previously added variables can be removed from the **Diagram** window using the **Diagram parameters** dialog box (see **Section 4.2.4.1**).

❖ **To remove a variable from a "Diagram" window:**

1. Double-click anywhere in the **Diagram** window, or
   Right-click anywhere in the **Diagram** window and choose **Parameters…** from the popup menu.
   The **Diagram parameters** dialog box will appear as shown in **Figure 184**.

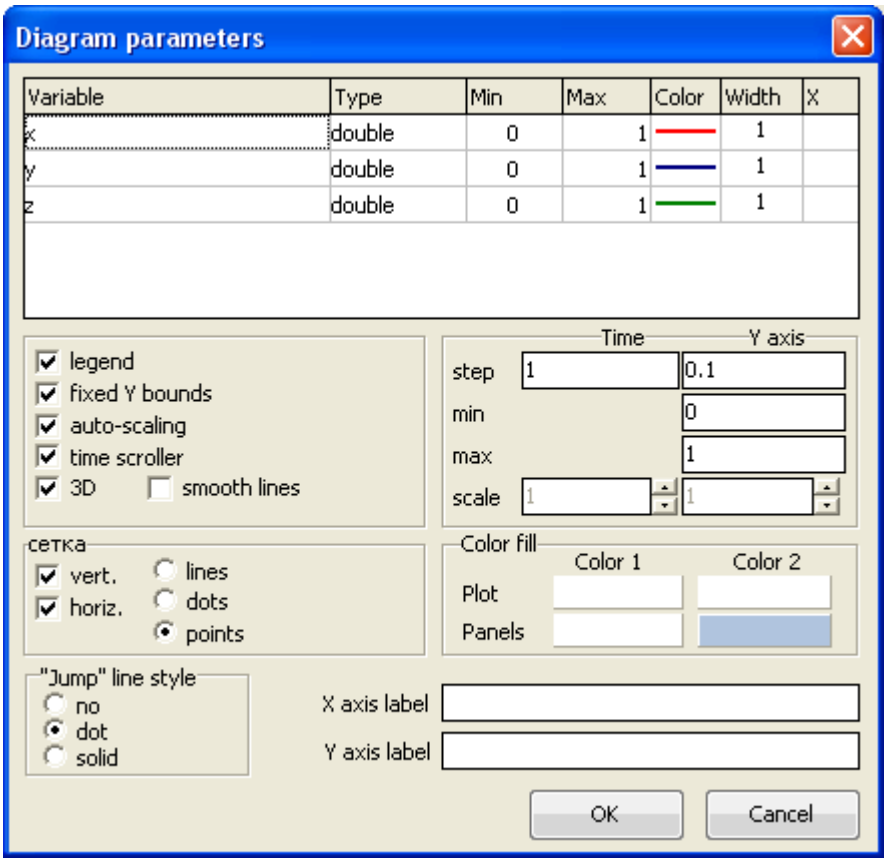


Figure 184 The "Diagram parameters" dialog box

2. Right click the variable in the list at the top and choose **Delete…** from the popup menu.
   A confirmation message box will appear.
3. Click **Yes**.
4. Click **OK** in the **Diagram parameters** dialog box.

The **Diagram** window plots `double`, `integer`, `short`, and `byte` values according to common plotting conventions. Different plots can either share a single y-axis or each use its own y-axis range and scale as shown in **Figure 185**.

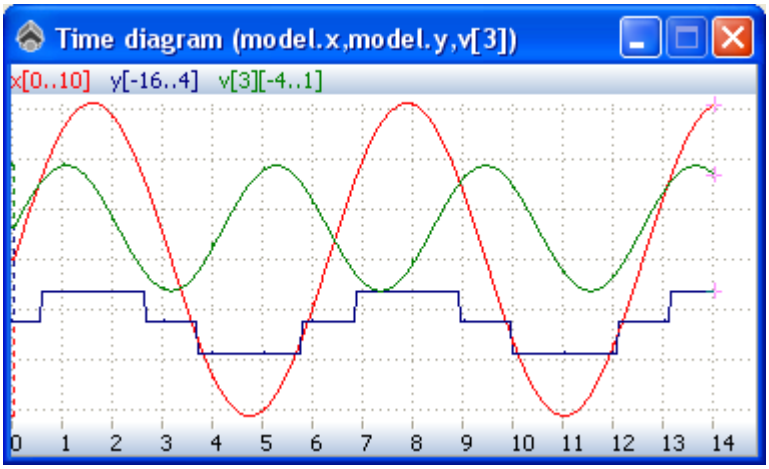


Figure 185 Different plots each using its own y-axis range and scale

Enumeration values are plotted so that each enumeration literal is assigned a value which is one grid step greater than the adjacent lower value and one grid step less than the adjacent upper value, see **Figure 186** for example.

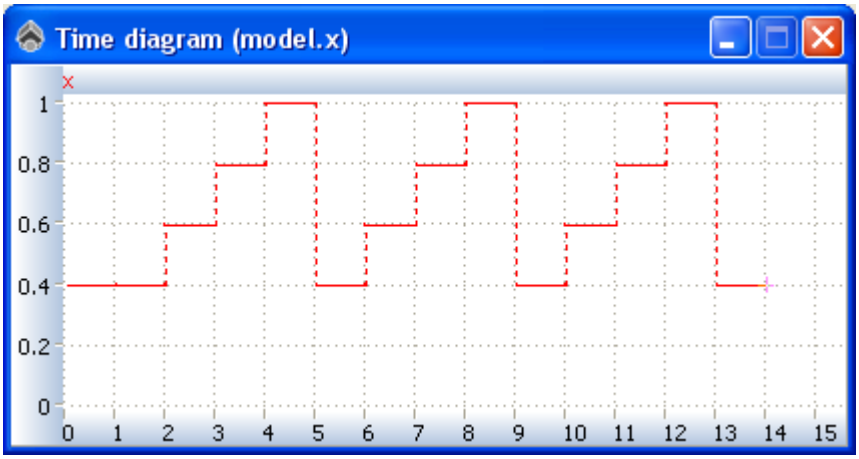


Figure 186 An enumeration variable plotted in the "Diagram" window

Signal variables are plotted so that each signal occurrence is denoted by a vertical stroke, as shown in **Figure 187**.

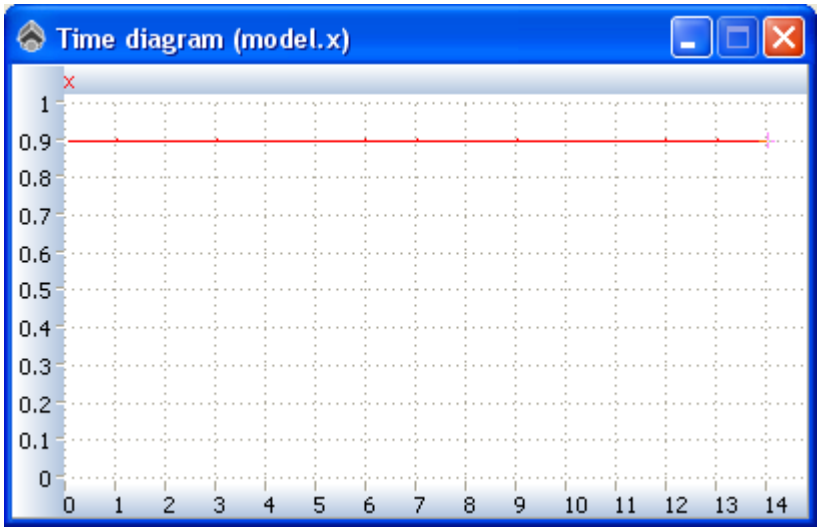


Figure 187 A signal variable plotted in the "Diagram" window

Every new **Diagram** window is configured to display a time diagram. If there is more than one variable on a time diagram, the diagram can at any time be converted to a phase diagram and back using the **Diagram parameters** dialog box. For more information, see **Section 4.2.4.1**.

The contents of a **Diagram** window can at any time be cleared, printed, or saved to an image or text file.

❖ **To clear a "Diagram" window:**

1. Right-click anywhere in the **Diagram** window and choose **Clear** from the popup menu.

❖ **To print a "Diagram" window:**

1. Right-click anywhere in the **Diagram** window and choose **Print…** from the popup menu.
   The standard **Print** dialog box will appear as shown in **Figure 188**.

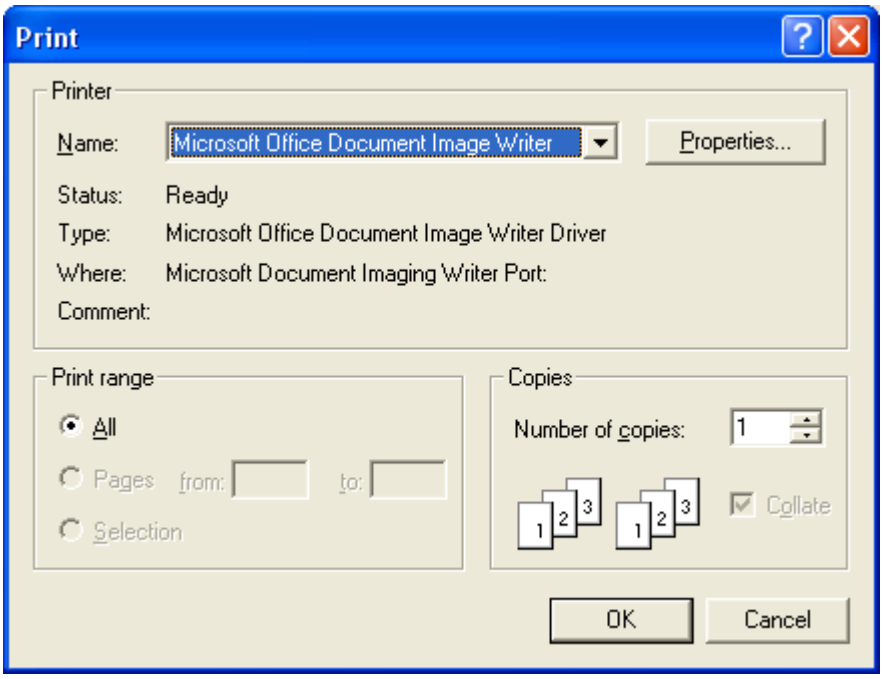


Figure 188 The "Print" dialog box

2. Adjust printing properties, if necessary, and click **OK**.

❖ **To save the contents of a "Diagram" window to a file as an image:**

1. Right-click anywhere in the **Diagram** window and choose **Save image…** from the popup menu.
   The **Save As** dialog box will appear.
2. Browse to the folder where you want to save the image.
3. Type the file name in the **File name** field.
4. Click **Save**.

❖ **To save the contents of a "Diagram" window to a file as a text table:**

1. Right-click anywhere in the **Diagram** window and choose **Export…** from the popup menu.
   The **Diagram data export** dialog box will appear as shown in **Figure 189**.

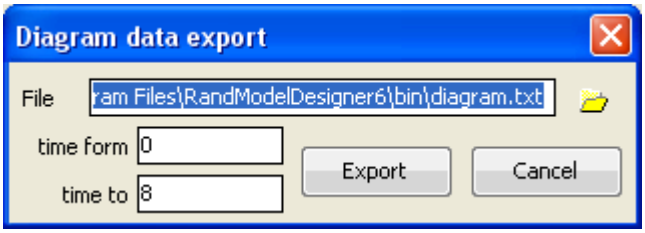


Figure 189 The "Diagram data export" dialog box

2. Type the path to the destination file in the **File** field, or click the button and specify the destination file in the dialog box that appears.
3. Specify the start and end time of the time interval you want to save in the **time from** and **time to** fields, respectively.
4. Click **Export**.
   The resulting file will be opened in the default text editor as shown in **Figure 190**.

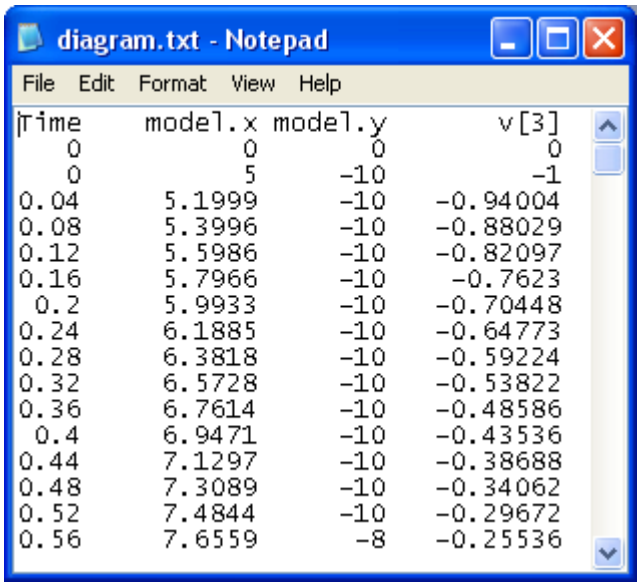


Figure 190 Export result

The **Diagram** window can be customized using the **Diagram parameters** dialog box.

4.2.4.1 The "Diagram parameters" Dialog Box

❖ **To open the "Diagram parameters" dialog box:**

1. Double-click anywhere in the **Diagram** window, or
   Right-click anywhere in the **Diagram** window and choose **Parameters…** from the popup menu.
   The **Diagram parameters** dialog box will appear as shown in **Figure 191**.

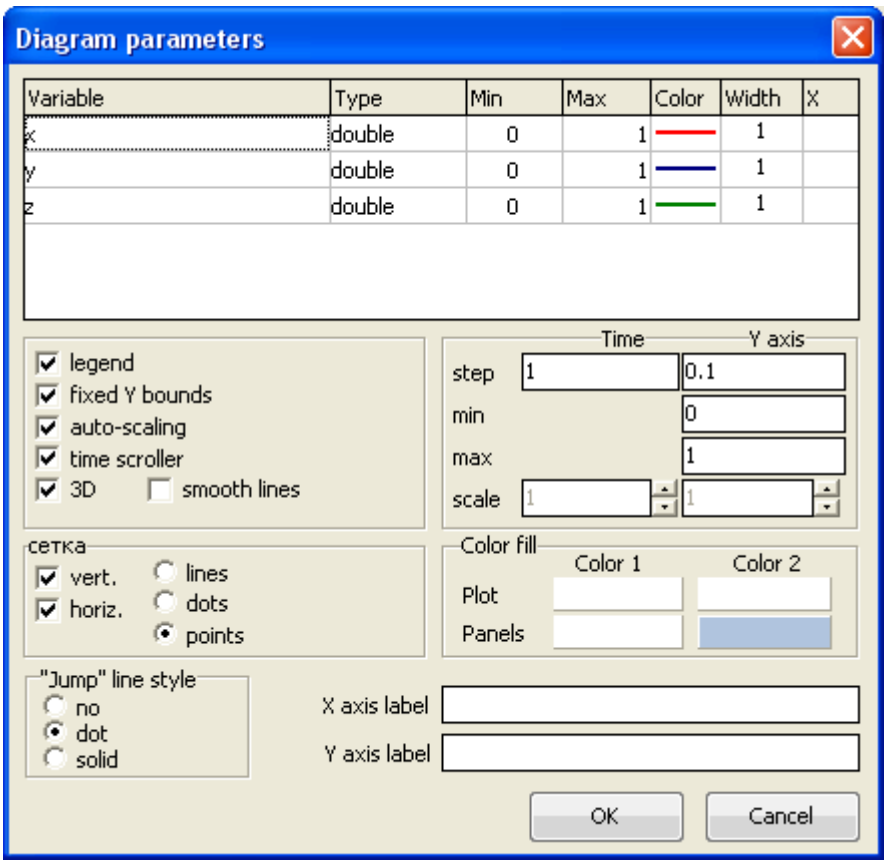


Figure 191 The "Diagram parameters" dialog box

At the top of the **Diagram parameters** dialog box, there is a list displaying the variables added to the **Diagram** window. The list has the following columns.

Variable – This read-only column shows the variable names.

Type – This read-only column shows the variable types.

Min – This column allows the user to specify the minimum value for each variable.

Max – This column allows the user to specify the maximum value for each variable.

Color – This column allows the user to choose the color used to plot each variable.

Width – This column allows the user to specify the line width used to plot each variable.

X – This column allows the user to choose the variable associated with the x-axis – the *x-axis variable*. The selected variable, if any, is denoted by the "+" sign shown in this column. At most one variable can be selected. If there is a variable denoted by the "+" sign, it means that the **Diagram** window displays a phase diagram and has the x-axis associated with the denoted variable: in this case, the **Diagram** window is said to be in the phase diagram mode. Otherwise, the **Diagram** window displays a time diagram and is said to be in the time diagram mode.

❖ **To modify the "Min", "Max", or "Width" attribute of a variable:**

1. Double-click the corresponding cell.
   An in-place editing field will appear.
2. Type the new value.
3. Hit *Enter* or click anywhere outside the editing field.

❖ **To modify the color used to plot a variable:**

1. Double-click the **Color** field of the variable.
   The standard **Color** dialog box will appear as shown in **Figure 192**.

Figure 192 The "Color" dialog box

2. Select the desired color and click **OK**.

❖ **To select or deselect the x-axis variable:**

1. Double-click the **X** field of the variable you want to select or deselect.

Besides the list of variables, the **Diagram parameters** dialog box contains the following elements:

legend – This checkbox shows/hides the legend at the top of the **Diagram** window.

fixed Y bounds – This checkbox enables/disables sharing of the y-axis by all plots.

auto-scaling – This checkbox enables/disables automatic scaling for the y-axis in the time diagram mode and for both axes in the phase diagram mode.

time scroller – This checkbox enables/disables scrolling along the x-axis in the time diagram mode.

3D – This checkbox enables/disables 3D display of empty areas around the chart.

smooth lines – This checkbox enables/disables line smoothing.

grid – This group of checkboxes and options configures the grid. The **vert.** checkbox shows/hides vertical grid lines. The **horiz.** checkbox shows/hides horizontal grid lines. The **lines** option selects solid grid lines. The **dots** option selects dashed grid lines. The **points** option selects dotted grid lines.

"Jump" line style – This group of options specifies how the **Diagram** window handles value gaps that occur when variables change discretely due to model state changes. **no** indicates that gaps remain open. **dots** indicates that gaps are closed with dotted lines. **solid** indicates that gaps are closed with normal solid lines.

step – These two fields, if present, specify the grid step along the x-axis (the first field) and the y-axis (the second field).

min – These two fields, if present, specify the minimum value along the x-axis (the first field) and the y-axis (the second field).

max – These two fields, if present, specify the maximum value along the x-axis (the first field) and the y-axis (the second field).

scale – These two fields, if present, specify the scaling factor for the x-axis (the first field) and the y-axis (the second field).

Color fill – This group of color selectors specifies the fill colors. The **Plot** colors are used to fill the plot area. The **Panels** colors are used to fill the rest of the diagram. In each row, **Color 1** specifies the starting color of the linear gradient while **Color 2** specifies the ending color.

X axis label – This field specifies the label displayed along the x-axis.

Y axis label – This field specifies the label displayed along the y-axis.

OK – This button applies the current settings and closes the **Diagram parameters** dialog box.

Cancel – This button closes the **Diagram parameters** dialog box without applying the current settings.

4.2.5 The "Histogram" Window

The **Histogram** window is intended for displaying the contents of a vector variable graphically.

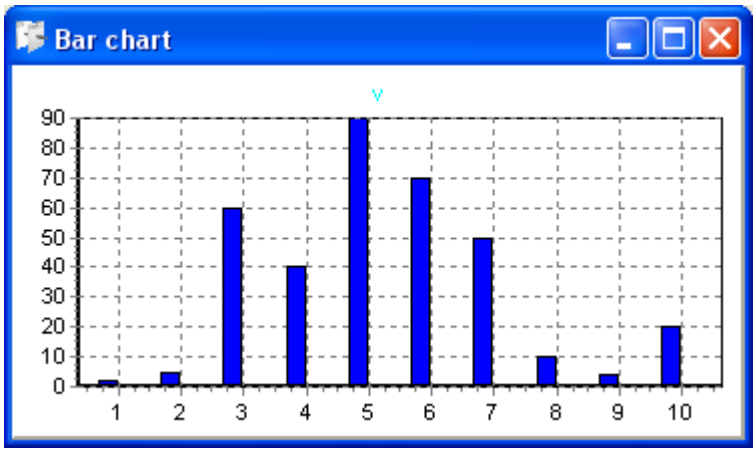


Figure 193 The "Histogram" window

A typical **Histogram** window is shown in **Figure 193**. The **Histogram** window displays a vector as a sequence of bars or other objects (see below), so that each object represents an individual vector element while the size/and or position of each object represents the value of the corresponding element.

One **Histogram** window can display the contents of one vector. The user can open as many **Histogram** windows as necessary.

❖ **To open a new "Histogram" window:**

1. Click **New histogram** on the main toolbar, or
   Choose **Window | New histogram** from the main menu.
   A new **Histogram** window will appear as shown in **Figure 194**.
   The window now needs to be associated with a vector variable.

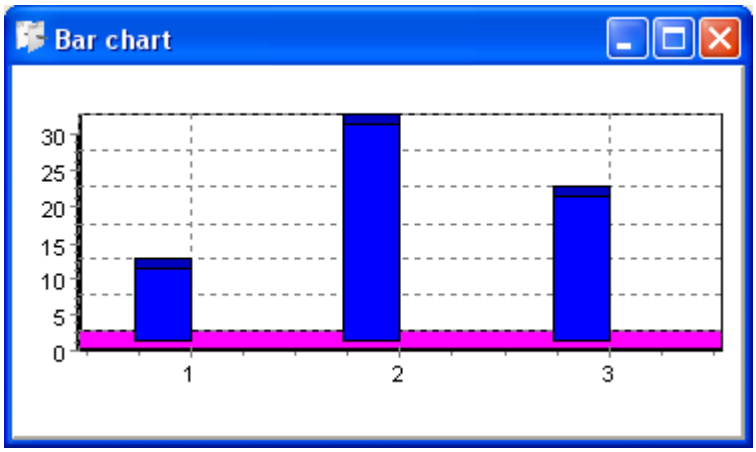


Figure 194 New "Histogram" window

❖ **To associate a "Histogram" window with a vector variable:**

1. Drag the variable from a **Variables** or **Structure** window and drop it onto the **Histogram** window.
   The **Histogram** window will start displaying the name and the contents of the associated variable.

The **Histogram** window supports the following view modes:

Vertical bars, see **Figure 195**

Horizontal bars, see **Figure 196**

Points, see **Figure 197**

Line, see **Figure 198**

Pie, see **Figure 199**

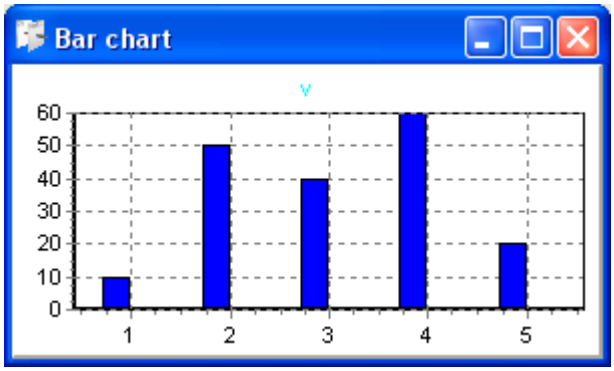


Figure 195 Vertical bars

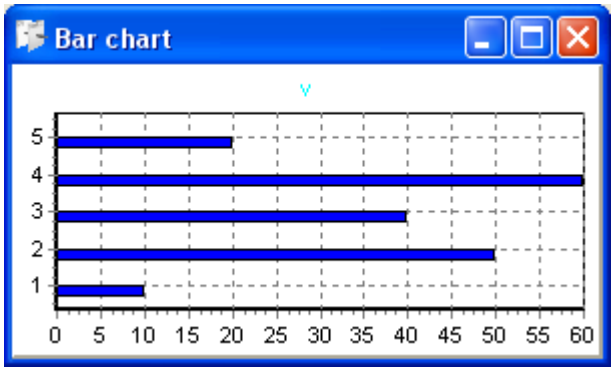


Figure 196 Horizontal bars

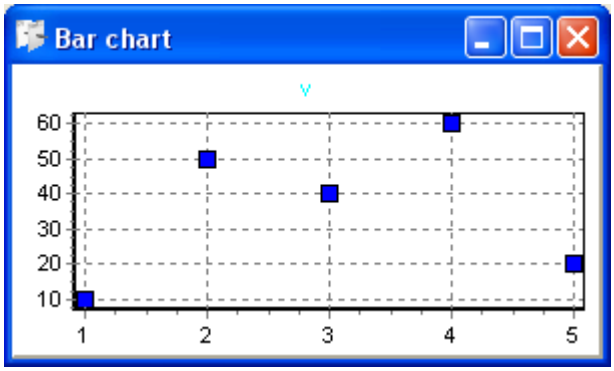


Figure 197 Points

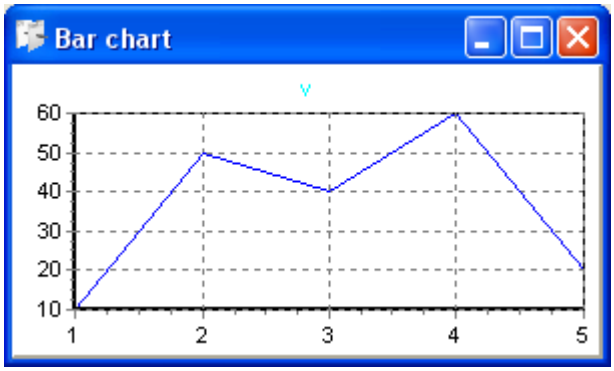


Figure 198 Line

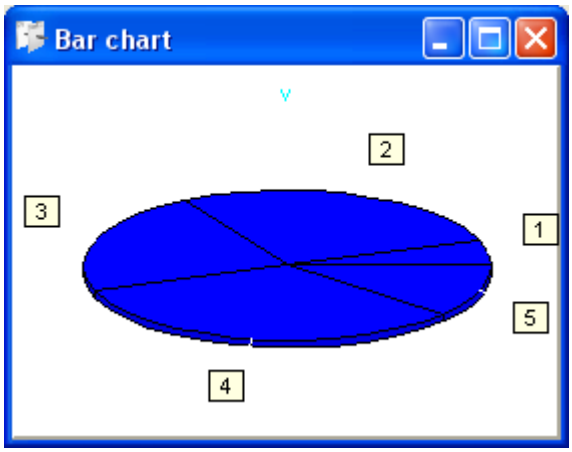


Figure 199 Pie

❖ **To change the view mode:**

1. Right-click anywhere in the **Histogram** window and select the desired mode in the **Mode** submenu of the popup menu.

All other properties of the **Histogram** window can also be configured via its context menu. The properties include the title displayed at the top of the window, 3D display flag, colors used to draw the diagram, background filling, x-axis labels, and minimum value for the y-axis.

❖ **To modify the title of a "Histogram" window:**

1. Right-click anywhere in the **Histogram** window and choose **Title…** from the popup menu.
   The **Title** dialog box will appear as shown in **Figure 200**.

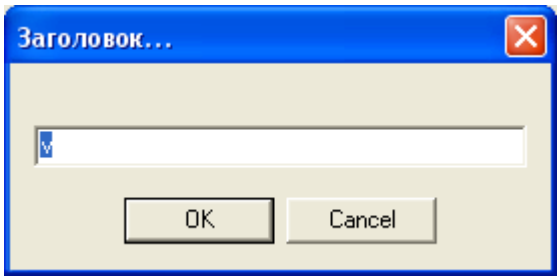


Figure 200 The "Title" dialog box

2. Type the new title.
3. Click **OK**.

❖ **To toggle 3D display:**

1. Right-click anywhere in the **Histogram** window and choose **3D** from the popup menu.

❖ **To modify the color of objects (bars, pie slices, points, or lines) displayed in a "Histogram" window:**

1. Right-click anywhere in the **Histogram** window and choose **Color | Elements…** from the popup menu.
   The standard **Color** dialog box will appear as shown in **Figure 201**.

Figure 201 The "Color" dialog box

2. Select the desired color and click **OK**.

❖ **To modify the title color:**

1. Right-click anywhere in the **Histogram** window and choose **Color | Title…** from the popup menu.
   The standard **Color** dialog box will appear as shown in **Figure 201**.

2. Select the desired color and click **OK**.

❖ **To modify the color of 3D surfaces:**

1. Right-click anywhere in the **Histogram** window and choose **Color | 3D Surfaces…** from the popup menu.
   The standard **Color** dialog box will appear as shown in **Figure 201**.

2. Select the desired color and click **OK**.

❖ **To modify the background filling:**

1. Right-click anywhere in the **Histogram** window and choose **Background | Starting color…** from the popup menu.
   The standard **Color** dialog box will appear as shown in **Figure 201**.

2. Select the desired starting color for linear gradient filling and click **OK**.

3. Right-click anywhere in the **Histogram** window and choose **Background | Ending color…** from the popup menu.
   The standard **Color** dialog box will appear again.

4. Select the desired ending color for linear gradient filling and click **OK**.

5. To toggle the gradient direction between horizontal and vertical, right-click anywhere in the **Histogram** window and choose **Background | Vertical gradient** from the popup menu.

❖ **To modify the x-axis labels:**

1. To modify the labeling mode, right-click anywhere in the **Histogram** window and choose either **No**, **Integer**, or **Real** from the **X-axis labels | Mode** submenu of the popup menu. **No** indicates that no labels will be shown. **Integer** enables integer labels. **Real** enables real number labels.

2. To modify the value of the first label, right-click anywhere in the **Histogram** window and choose **X-axis labels | First label…** from the popup menu.
   The **First Label** dialog box will appear as shown in **Figure 202**.
   In the **First Label** dialog box, type the new value for the first label and click **OK**.

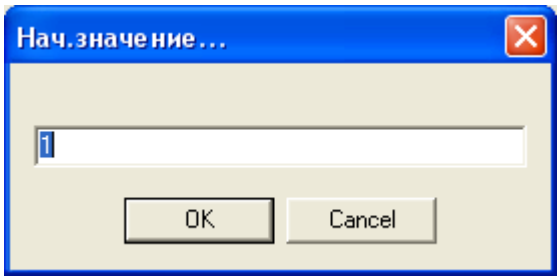


Figure 202 The "First Label" dialog box

3. To modify the increment between labels, right-click anywhere in the **Histogram** window and choose **X-axis labels | Increment…** from the popup menu.
   The **Increment** dialog box will appear as shown in **Figure 203**.
   In the **Increment** dialog box, type the new value for the increment and click **OK**.

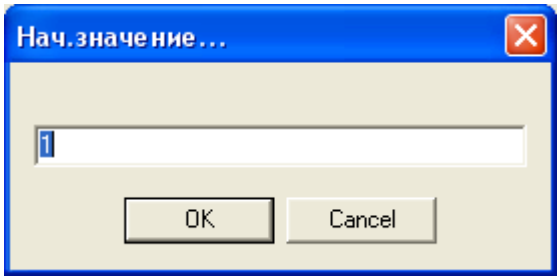


Figure 203 The "Increment" dialog box

4. To modify the number of decimal places in real number labels, right-click anywhere in the **Histogram** window and choose **X-axis labels | Precision…** from the popup menu.
   The **Precision** dialog box will appear as shown in **Figure 204**.
   In the **Precision** dialog box, type the new value and click **OK**.

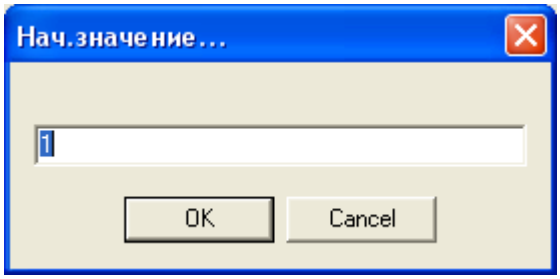


Figure 204 The "Precision" dialog box

❖ **To modify the minimum y-axis value:**

1. Right-click anywhere in the **Histogram** window and choose **Minimum y-axis value…** from the popup menu.
   The **Minimum Y-Axis Value** dialog box will appear as shown in **Figure 205**.

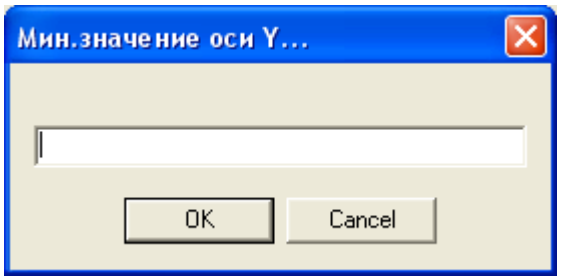


Figure 205 The "Minimum Y-Axis Value" dialog box

2. Type the new value and click **OK**.

The contents of a **Histogram** window can at any time be saved to a file as an image or printed.

❖ **To save the contents of a "Histogram" window to a file as an image:**

1. Right-click anywhere in the **Histogram** window and choose **Save image…** from the popup menu.
   The **Save As** dialog box will appear.
2. Browse to the folder where you want to save the image.
3. Type the file name in the **File name** field.
4. Click **Save**.

❖ **To print the contents of a "Histogram" window:**

1. Right-click anywhere in the **Histogram** window and choose **Print…** from the popup menu.
   The standard **Print** dialog box will appear as shown in **Figure 206**.

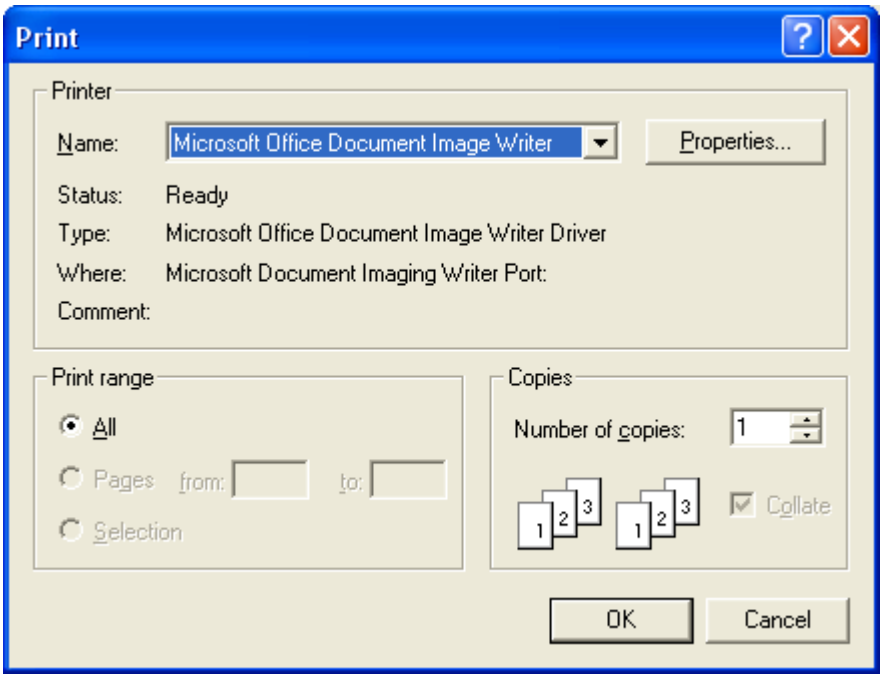


Figure 206 The "Print" dialog box

2. Adjust printing properties, if necessary, and click **OK**.

4.2.6 The "2D Animation" Window

As follows from its name, the **2D Animation** window is intended for two-dimensional animation. The **2D Animation** window provides a two-dimensional scene that can be populated with standard animation components such as indicators and sprites. Components can be associated with model variables so that when model variables change, components move and change their appearance accordingly, thereby animating the model. The user can open and populate as many **2D Animation** windows as necessary. An example of the **2D Animation** window is shown in **Figure 207**.

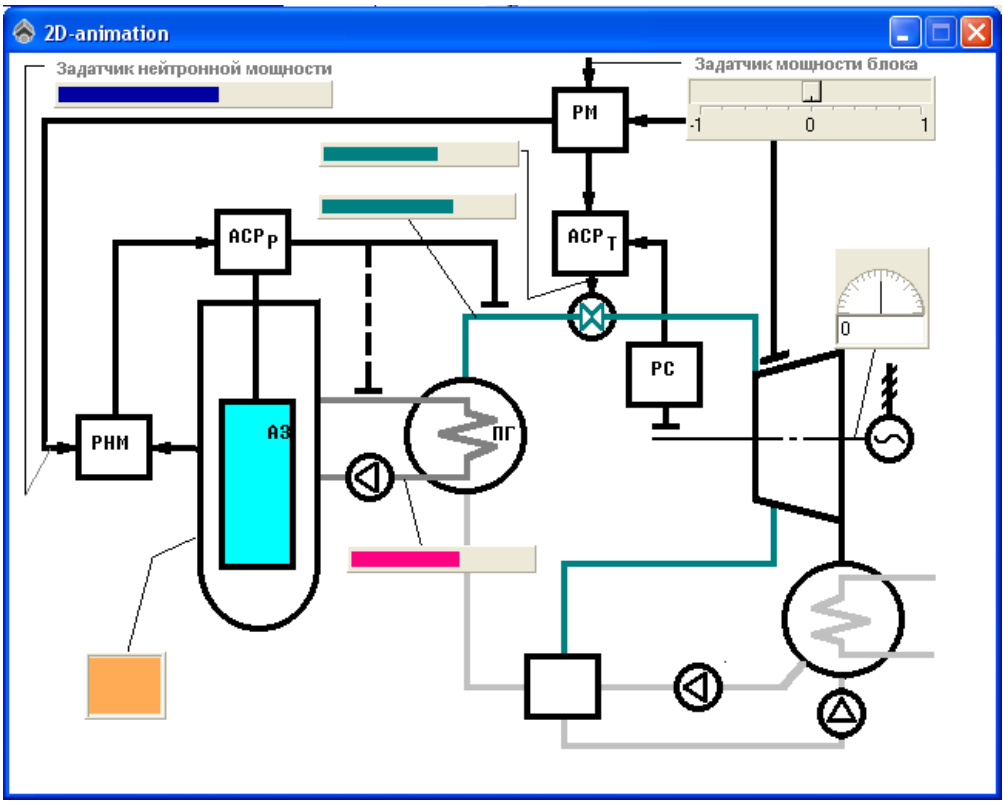


Figure 207 “2D Animation” window example

❖ **To open a new “2D Animation” window:**

1. Click **New 2D animation** on the main toolbar, or
   Choose **Window | New 2D animation** from the main menu.
   A new **2D Animation** window will appear as shown in **Figure 208**.

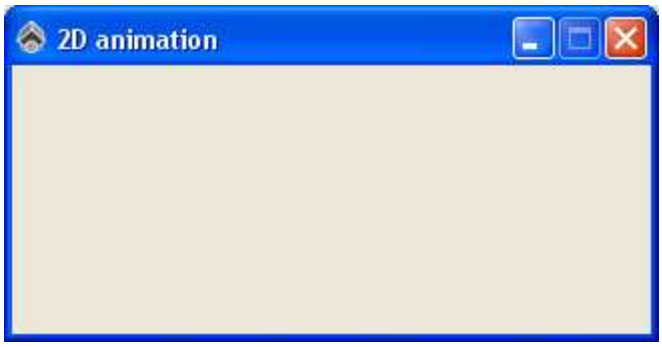


Figure 208 New “2D Animation” window

The user populates the **2D Animation** window by dragging and dropping components from the **Standard 2D components** panel.

4.2.6.1 The “Standard 2D components” Panel

The **Standard 2D components** panel, see **Figure 209**, provides a palette of components that can be dropped onto a **2D Animation** or **Structure** window.

Figure 209 The “Standard 2D components” panel

❖ **To open the "Standard 2D components" panel:**

1. Click **Standard 2D components** on the main toolbar, or
   Choose **Service | Standard 2D components** from the main menu.

The panel provides the following components:

– dancer

 – continuous linear indicator

 – discrete linear indicator

– slider

– color indicator

– sprite

– diagram

– digital indicator

– knob

– button

4.2.6.2 Using the "2D Animation" Window

❖ **To add a component to a "2D Animation" window:**

1. Drag the component from the **Standard 2D components** panel and drop it onto the **2D Animation** window.

❖ **To associate a component with a variable:**

1. Drag the variable from a **Variables** or **Structure** window and drop it onto the component in the **2D Animation** window.

❖ **To move a component:**

1. Drag the component in the desired direction.

❖ **To resize a component:**

1. Drag any border of the component in the desired direction.

❖ **To delete a component from a "2D Animation" window:**

1. Right-click the component and choose **Close** from the popup menu.

❖ **To set or change the background image of a "2D Animation" window:**

1. Right-click anywhere in the **2D Animation** window and choose **Background…** from the popup menu.
   The **Open** dialog box will appear.
2. Select the file you want to use as a background and click **Open**.

❖ **To clear the background:**

1. Right-click anywhere in the **2D Animation** window and choose **Clear background** from the popup menu.

❖ **To modify the window title:**

1. Right-click anywhere in the **2D Animation** window and choose **Window caption…** from the popup menu.
   The **Edit window caption** dialog box will appear as shown in **Figure 210**.

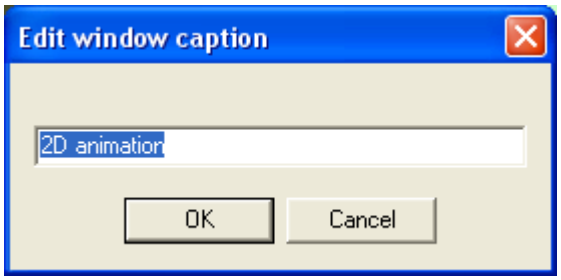


Figure 210 The "Edit window caption" dialog box

2. Type the new window title.
3. Click **OK**.

❖ **To add a text label:**

1. Right-click where you want to place a text label and choose **New label…** from the popup menu.
   The **Enter the label text** dialog box will appear as shown in **Figure 211**.

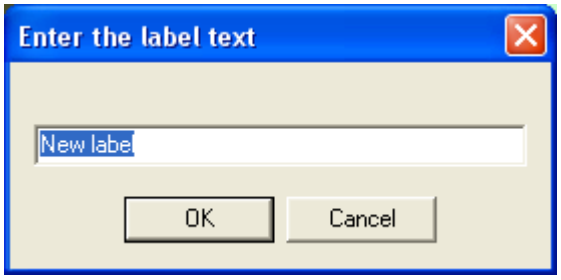


Figure 211 The "Enter the label text" dialog box

2. Type the label text and click **OK**.

❖ **To toggle the edit mode:**

1. Right-click anywhere in the **2D Animation** window and choose **Editing** from the popup menu.
   When a **2D Animation** window is not in the edit mode, its contents cannot be modified.

The properties of components added to a **2D Animation** window can be modified via their context menus. The following subsections describe the supported component types and their context menu items.

4.2.6.3 Dancer

This component provides the following context menu items:

Min – Allows to specify the minimum value displayed by the dancer.

Max – Allows to specify the maximum value displayed by the dancer.

Style – Allows to convert the dancer to a component of any other type.

Value – Shows/hides the box displaying the associated variable value.

Value format – Allows to specify the floating-point number display format.

Show scale – Shows/hides the scale labels.

Auto-scaling – Toggles automatic scaling.

Position – Allows to specify the position and size of the component. The position is relative to the upper-left corner of the **2D Animation** window.

4.2.6.4 Linear Indicator (Continuous or Discrete)

This component provides the following context menu items:

Min – Allows to specify the minimum value displayed by the indicator.

Max – Allows to specify the maximum value displayed by the indicator.

Style – Allows to convert the indicator to a component of any other type.

Vertical – Toggles between vertical and horizontal orientation.

Value – Shows/hides the box displaying the associated variable value.

Color – Allows to specify the indicator color.

Value format – Allows to specify the floating-point number display format.

Show scale – Shows/hides the scale labels.

Auto-scaling – Toggles automatic scaling.

Position – Allows to specify the position and size of the component. The position is relative to the upper-left corner of the **2D Animation** window.

4.2.6.5 Slider

This component provides the following context menu items:

Min – Allows to specify the minimum value displayed by the slider.

Max – Allows to specify the maximum value displayed by the slider.

Style – Allows to convert the slider to a component of any other type.

Vertical – Toggles between vertical and horizontal orientation.

Value – Shows/hides the box displaying the associated variable value.

Value format – Allows to specify the floating-point number display format.

Show scale – Shows/hides the scale labels.

Auto-scaling – Toggles automatic scaling.

Position – Allows to specify the position and size of the component. The position is relative to the upper-left corner of the **2D Animation** window.

4.2.6.6 Color Indicator

This component provides the following context menu items:

Min – Allows to specify the minimum value displayed by the indicator.

Max – Allows to specify the maximum value displayed by the indicator.

Style – Allows to convert the indicator to a component of any other type.

Value – Shows/hides the box displaying the associated variable value.

Value format – Allows to specify the floating-point number display format.

Show scale – Shows/hides the scale labels.

Auto-scaling – Toggles automatic scaling.

Color min – Allows to choose the color that corresponds to the minimum value.

Color max – Allows to choose the color that corresponds to the maximum value.

Position – Allows to specify the position and size of the component. The position is relative to the upper-left corner of the **2D Animation** window.

4.2.6.7 Sprite

The sprite component represents a rectangular area that can move around the animation and display one of the pre-configured images. Every sprite can have up to three variables associates with it. The x-coordinate of the sprite can be associated with one variable. The y-coordinate of the sprite can be associated with another variable. The index of the currently displayed image can be associated with the third variable.

❖ **To associate a variable with a sprite:**

1. Drag the variable from a **Variables** or **Structure** window and drop it onto the sprite in the **2D Animation** window.
   The **Use Variable As** dialog box will appear as shown in **Figure 212**.

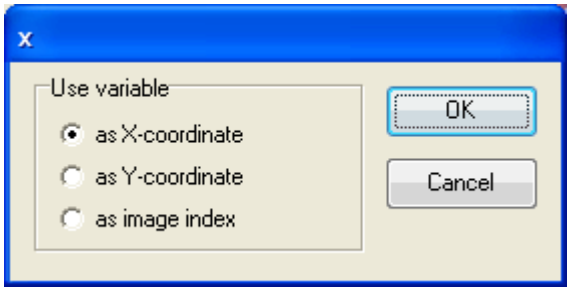


Figure 212 The "Use Variable As" dialog box

2. To associate the variable with the x-coordinate of the sprite, choose the **X-coordinate** option. To associate the variable with the y-coordinate, choose the **Y-coordinate** option. To associate the variable with the index of the currently displayed image, choose the **Image index** option.

3. Click **OK**.

The image or images that a sprite will display can be added to the sprite via the **Animation parameters** dialog box (see **Section 4.2.6.7.1**).

❖ **To add one or more images to a sprite:**

1. Right-click the sprite and choose **Parameters...** from the popup menu.
   The **Animation parameters** dialog box will appear as shown in **Figure 213**.

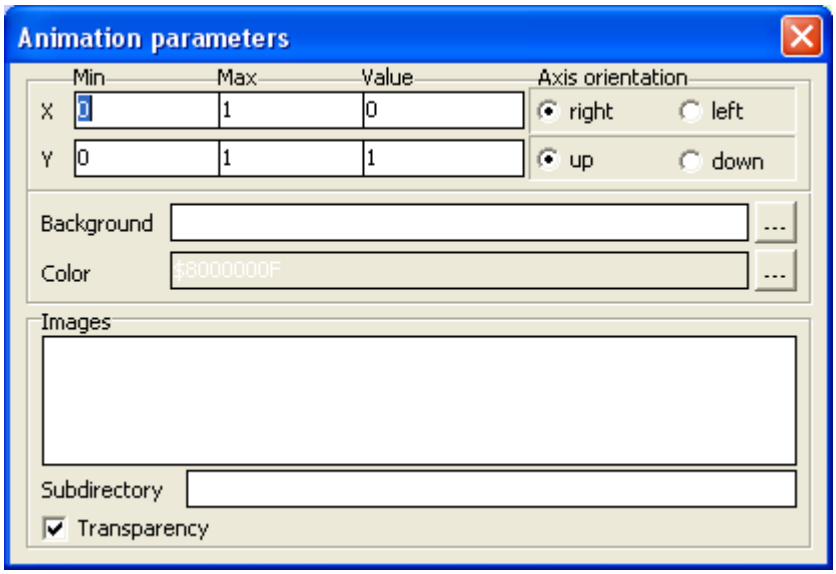


Figure 213 The "Animation parameters" dialog box

2. Right-click anywhere in the **Images** list and choose **Add…** from the popup menu.
   The **Open** dialog box will appear.
3. Select the image file you want to add and click **Open**.
   The file will be added to the **Images** list.
4. If necessary, repeat the above two steps to add more images.
5. Set the checkbox next to the image that should be displayed by default.
6. Close the **Animation parameters** dialog box.

Every sprite has the associated movement area that can be toggled on and off. When toggled on, the movement area is displayed as a hatched rectangle inside the sprite. When toggled on, the movement area can be moved and resized.

❖ **To toggle the movement area of a sprite:**

1. Right-click the sprite and choose **Track rectangle** from the popup menu.

❖ **To move the movement area:**

1. Drag the movement area in the desired direction.

❖ **To resize the movement area:**

1. Drag any border of the movement area in the desired direction.

Other sprite properties can be customized using the **Animation parameters** dialog box.

4.2.6.7.1 The "Animation parameters" Dialog Box

❖ **To open the "Animation parameters" dialog box for a sprite:**

1. Right-click the sprite and choose **Parameters…** from the popup menu.
   The **Animation parameters** dialog box will appear as shown in **Figure 213**.

The **Animation parameters** dialog box contains the following elements:

X – These three fields specify the minimum (**Min**), maximum (**Max**), and initial (**Value**) x-coordinate of the movement area.

Y – These three fields specify the minimum (**Min**), maximum (**Max**), and initial (**Value**) y-coordinate of the movement area.

right, **left** – These options specify the x-axis orientation for the movement area.

up, **down** – These options specify the y-axis orientation for the movement area.

Background – This field specifies the path to the background image.

Color – This field specifies the background fill color.

Images – This area shows the list of image files associated with a sprite. The default image is denoted by a checked checkbox. The list provides a context menu containing the following items: **Add…**, **Insert…**, **Delete**. **Add…** adds an image to the end of the list. **Insert…** inserts an image before the currently selected image. **Delete** deletes the currently selected image.

Transparency – This checkbox enables/disables transparency. If this checkbox is checked, the color of the lower-left pixel of every image is rendered as transparent for that image. Otherwise, all images are rendered opaque.

4.2.6.8 Diagram

The diagram component represents a **Diagram** window embedded into a **2D Animation** window, functioning just like a standalone **Diagram** window. For more information, see **Section 4.2.4**.

4.2.6.9 Digital Indicator

This component provides the following context menu items:

Style – Allows to convert the indicator to a component of any other type.

Value – Shows/hides the box displaying the associated variable value.

Font – Allows to specify the font used to display the indicator value.

Color – Allows to specify the indicator color.

Value format – Allows to specify the floating-point number display format.

Position – Allows to specify the position and size of the component. The position is relative to the upper-left corner of the **2D Animation** window.

4.2.6.10 Knob

This component provides the following context menu items:

Min – Allows to specify the minimum value displayed by the knob.

Max – Allows to specify the maximum value displayed by the knob.

Style – Allows to convert the knob to a component of any other type.

Value – Shows/hides the box displaying the associated variable value.

Value format – Allows to specify the floating-point number display format.

Auto-scaling – Toggles automatic scaling.

Position – Allows to specify the position and size of the component. The position is relative to the upper-left corner of the **2D Animation** window.

4.2.6.11 Button

This component provides the following context menu items:

Style – Allows to convert the button to a component of any other type.

Caption – Allows to specify the button label.

Fixation – Toggles the two-state button mode. In the two-state button mode, each click on a button toggles the button between two states: pressed and not pressed. In the normal push-button mode, the button returns to its un-pushed state automatically after every click.

Shortcut – Allows to assign a keyboard shortcut to the button.

Picture – Allows to specify the image to be displayed on the button.

Frame – Allows to specify the margin between the component (outer) border and the button (inner) border.

Font – Allows to choose the button font.

Color min – Allows to choose the starting color of the button's gradient fill.

Color max – Allows to choose the ending color of the button's gradient fill.

Position – Allows to specify the position and size of the component. The position is relative to the upper-left corner of the **2D Animation** window.

4.2.7 The "3D Animation" Window

The **3D Animation** window is intended for three-dimensional animation. The **3D Animation** window provides a three-dimensional scene that can be populated with standard 3D objects such as spheres, cylinders, and cones. Object properties can be associated with model variables so that when model variables change, objects move and change their appearance accordingly, thereby animating the model. The user can open and populate as many **3D Animation** windows as necessary. An example of the **3D Animation** window is shown in **Figure 214**.

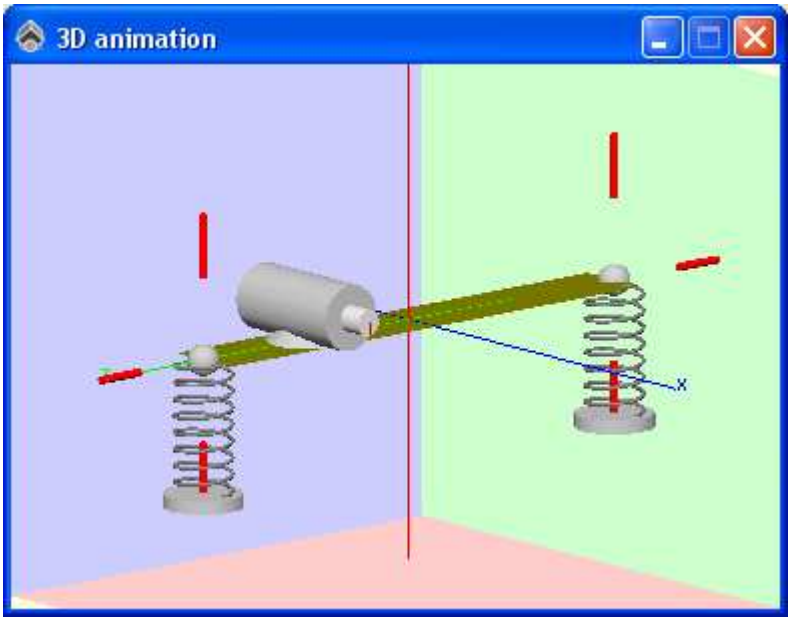


Figure 214 "3D Animation" window example

❖ **To open a new "3D Animation" window:**

1. Click **New 3D animation** on the main toolbar, or
   Choose **Window | New 3D animation** from the main menu.
   A new **3D Animation** window will appear as shown in **Figure 215**.

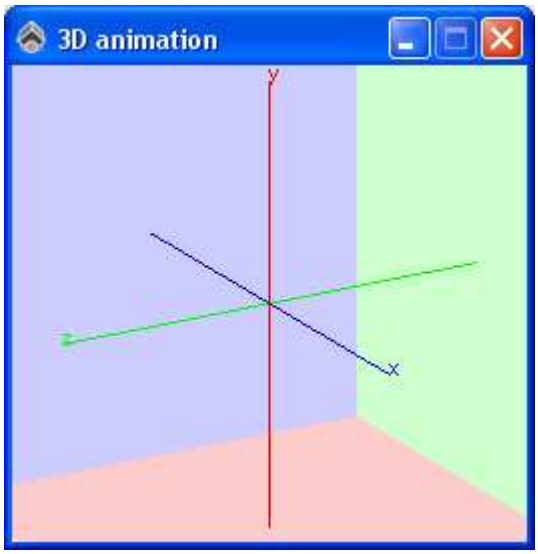


Figure 215 New "3D Animation" window

The user populates and customizes the **3D Animation** window using the **3D animation parameters** dialog box.

4.2.7.1 The "3D animation parameters" Dialog Box

❖ **To open the "3D animation parameters" dialog box:**

1. Double-click anywhere in the **3D Animation** window, or
   Right-click anywhere in the **3D Animation** window and choose **Parameters…** from the popup menu.
   The **3D animation parameters** dialog box will appear as shown in **Figure 216**.

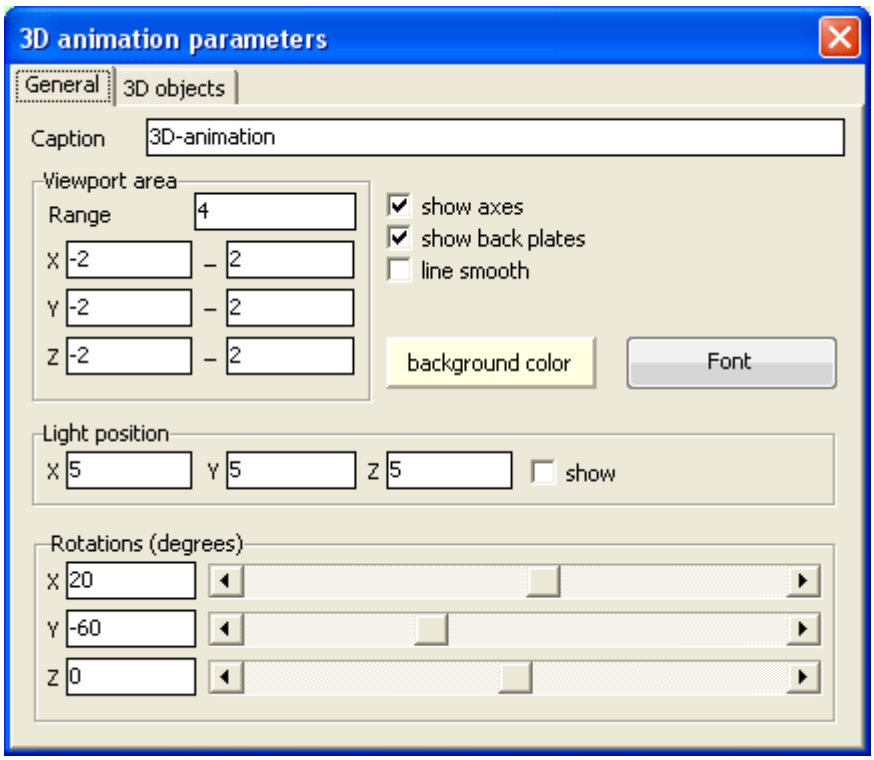


Figure 216 The "3D animation parameters" dialog box, the "General" tab

The **3D animation parameters** dialog box consists of two tabs: **General** and **3D objects**.

4.2.7.1.1 The “General” Tab

The **General** tab, see **Figure 216**, contains the following elements:

Caption – This field specifies the title of the **3D Animation** window.

Viewport area – This groups of fields defines the viewport area.

show axes – This checkbox shows/hides the axes.

show back plates – This checkbox shows/hides the background surfaces.

line smooth – This checkbox enables/disables line smoothing.

Light position – This group of elements defines the location of the light source. The **X**, **Y**, and **Z** fields specify the coordinates of the light source. The **show** checkbox shows/hides the light source marker.

Rotations – This group of elements specifies the angles of rotation of the animation around its axes. When all angles are zeros, the x-axis points right, the y-axis points up, and the z-axis points out of the screen, towards the viewer.

4.2.7.1.2 The “3D objects” Tab

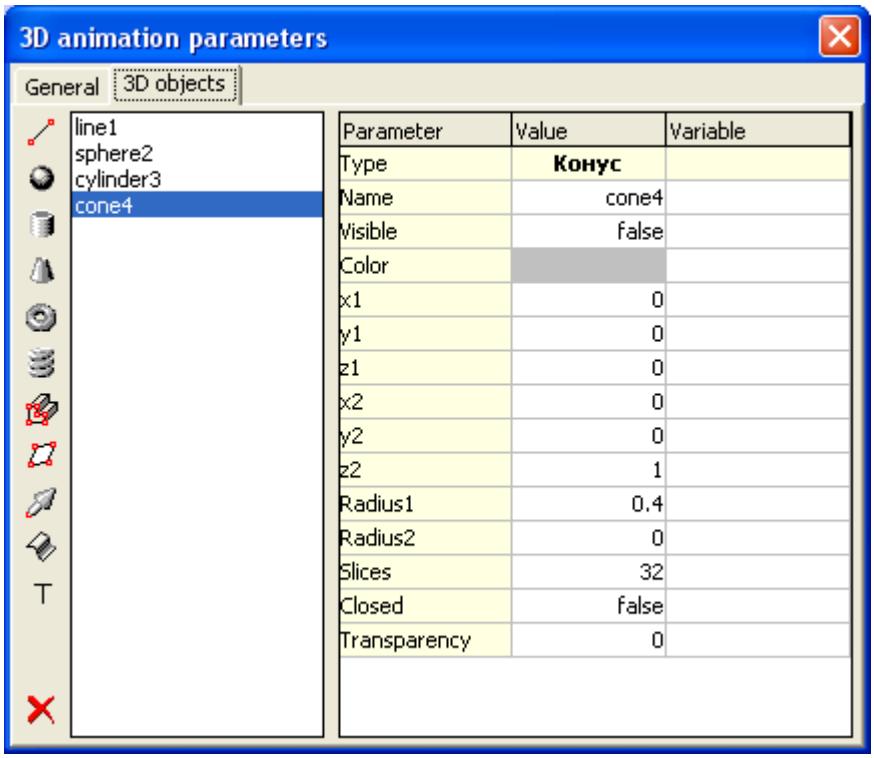


Figure 217 The “3D objects” tab

The 3D objects tab, see **Figure 217**, consists of the toolbar, object list, and property panel.

The toolbar provides the following buttons:

– adds a line segment.

– adds a sphere.

– adds a cylinder.

– adds a cone.

– adds a torus.

– adds a spring.

– adds a beam.

– adds a quadrangle.

– adds an arrow.

– adds a surface.

– adds a two-dimensional text label.

– deletes the currently selected object.

The object list displays all added objects and allows the user to select any one of them.

The property panel provides access to the properties of the currently selected object. The property panel has the following columns:

Parameter – This column displays the name of each property.

Value – This column displays and allows the user to modify the current value of each property. The current value may be either specified by the user or automatically set by the model to the current value of the associated variable, if any.

Variable – This column specifies the name of the variable associated with each property. The user can either enter variable names manually or drag variables from other windows and drop them onto fields in this column.

4.2.7.2 Using the "3D Animation" Window

❖ **To add an object to a "3D Animation" window:**

1. Open the **3D animation parameters** dialog box.
2. Activate the **3D objects** tab.
3. Click an appropriate button on the **3D objects** tab toolbar.

❖ **To remove an object from a "3D Animation" window:**

1. Open the **3D animation parameters** dialog box.
2. Activate the **3D objects** tab.
3. Select the object in the object list.
4. Click **Delete** on the **3D objects** tab toolbar.

❖ **To modify the current value of an object property:**

1. Open the **3D animation parameters** dialog box.
2. Activate the **3D objects** tab.
3. Select the object in the object list.
4. Click the **Value** field of the property in the property panel.
   An in-place editing field will appear.
5. Type the new value.
6. Hit *Enter* or click anywhere outside the editing field.

❖ **To associate a variable with an object property:**

1. Open the **3D animation parameters** dialog box.

2. Activate the **3D objects** tab.
3. Select the object in the object list.
4. Drag the variable from a **Variables** or **Structure** window and drop it onto the **Variable** field of the property in the property panel.

❖ **To configure the color of an object:**

1. Open the **3D animation parameters** dialog box.
2. Activate the **3D objects** tab.
3. Select the object in the object list.
4. Click the **Value** field of the **Color** property in the property panel.
5. If there is no variable associated with the **Color** property, the standard **Color** dialog box will appear.
   In the **Color** dialog box, choose the desired color and click **OK**.
6. If there is a variable associated with the **Color** property, the **Color Setup** dialog box will appear as shown in **Figure 218**.
   In the **Color Setup** dialog box, specify the minimum and maximum values of the associated variable in the **Minimum value** and **Maximum value** fields respectively, and choose the corresponding colors in the **Minimum color** and **Maximum color** fields. The color of the object will vary between the specified colors, proportionally to the current value of the associated variable.

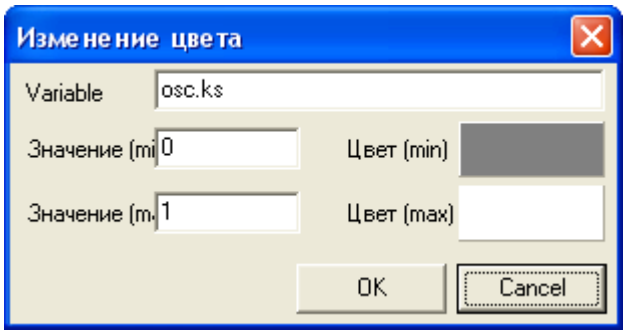


Figure 218 The "Color Setup" dialog box

❖ **To disassociate a variable from an object property:**

1. Open the **3D animation parameters** dialog box.
2. Activate the **3D objects** tab.
3. Select the object in the object list.
4. Click the **Variable** field of the property in the property panel.
   An in-place editing field will appear.
5. Hit *Delete*.

❖ **To rotate the scene within a "3D Animation" window:**

1. Drag the contents of the **3D Animation** window in the desired direction.

❖ **To zoom a "3D Animation window" in:**

1. Click anywhere in the **3D Animation** window while holding *Shift*.

❖ **To zoom a "3D Animation window" out:**

1. Click anywhere in the **3D Animation** window while holding *Control*.

❖ **To save the contents of a "3D Animation" window to a file as an image:**

1. Right-click anywhere in the **3D Animation** window and choose **Save image…** from the popup menu.
   The **Save As** dialog box will appear.
2. Browse to the folder where you want to save the image.
3. Type the file name in the **File name** field.
4. Click **Save**.

❖ **To record the animation and save it to an AVI file:**

1. Right-click anywhere in the **3D Animation** window and choose **Create AVI…** from the popup menu.
   A message explaining the recording process will appear as shown in **Figure 219**.

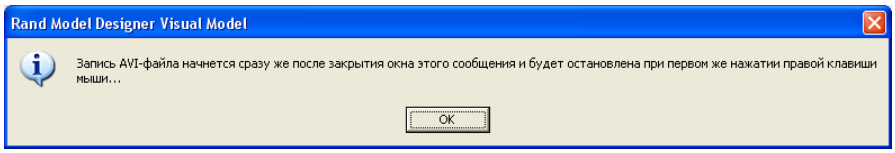


Figure 219 Recording start message

2. Click **OK**.
   The **3D Animation** window will start recording the animation.
3. To stop recording, right-click anywhere in the **3D Animation** window.
   The **Save As** dialog box will appear.
4. Browse to the folder where you want to save the recorded video.
5. Type the file name in the **File name** field.
6. Click **Save**.

The sections that follow describe the objects supported by the **3D Animation** window.

4.2.7.3 Line Segment

This object has the following properties:

Type – reflects the type of the object, read-only, cannot be associated with a variable.

Name – specifies the name of the object, cannot be associated with a variable.

Visible – determines whether the object is visible.

Color – specifies the color.

x1, **y1**, **z1** – specify the coordinates of the starting point.

x2, **y2**, **z2** – specify the coordinates of the ending point.

Width – specifies the line width in pixels.

4.2.7.4 Sphere

This object has the following properties:

Type – reflects the type of the object, read-only, cannot be associated with a variable.

Name – specifies the name of the object, cannot be associated with a variable.

Visible – determines whether the object is visible.

Color – specifies the color.

x1, **y1**, **z1** – specify the coordinates of the center of the sphere.

Radius – specifies the radius of the sphere.

Slices – specifies the number of segments used to model the sphere.

Transparency – specifies the object transparency, must be between 0 (fully opaque) and 1 (fully transparent).

4.2.7.5 Cylinder

This object has the following properties:

Type – reflects the type of the object, read-only, cannot be associated with a variable.

Name – specifies the name of the object, cannot be associated with a variable.

Visible – determines whether the object is visible.

Color – specifies the color.

x1, **y1**, **z1** – specify the coordinates of the center of the first base.

x2, **y2**, **z2** – specify the coordinates of the center of the second base.

Radius – specifies the radius of the cylinder.

Slices – specifies the number of segments used to model the cylinder.

Closed – specifies whether the bases are capped.

Transparency – specifies the object transparency, must be between 0 (fully opaque) and 1 (fully transparent).

4.2.7.6 Cone

This object has the following properties:

Type – reflects the type of the object, read-only, cannot be associated with a variable.

Name – specifies the name of the object, cannot be associated with a variable.

Visible – determines whether the object is visible.

Color – specifies the color.

x1, **y1**, **z1** – specify the coordinates of the center of the first base.

x2, **y2**, **z2** – specify the coordinates of the center of the second base.

Radius1 – specifies the radius of the first base.

Radius2 – specifies the radius of the second base.

Slices – specifies the number of segments used to model the cone.

Closed – specifies whether the bases are capped.

Transparency – specifies the object transparency, must be between 0 (fully opaque) and 1 (fully transparent).

4.2.7.7 Torus

This object has the following properties:

Type – reflects the type of the object, read-only, cannot be associated with a variable.

Name – specifies the name of the object, cannot be associated with a variable.

Visible – determines whether the object is visible.

Color – specifies the color.

x1, **y1**, **z1** – specify the coordinates of the center of the torus.

nX, **nY**, **nZ** – specify the normal vector that represents the axis of symmetry of the torus.

Radius – specifies the radius of the torus, that is, the distance from the center of the torus to the center of the tube.

Thickness – specifies the radius of the tube.

Slices – specifies the number of segments used to model the torus.

Transparency – specifies the object transparency, must be between 0 (fully opaque) and 1 (fully transparent).

4.2.7.8 Spring

This object has the following properties:

Type – reflects the type of the object, read-only, cannot be associated with a variable.

Name – specifies the name of the object, cannot be associated with a variable.

Visible – determines whether the object is visible.

Color – specifies the color.

x1, **y1**, **z1** – specify the coordinates of the starting point of the spring.

x2, **y2**, **z2** – specify the coordinates of the ending point of the spring.

Radius – specifies the radius of the spring cylinder.

Thickness – specifies the spring thickness.

Turns – specifies the number of turns.

Slices – specifies the number of segments used to model the spring.

Transparency – specifies the object transparency, must be between 0 (fully opaque) and 1 (fully transparent).

4.2.7.9 Beam

This object has the following properties:

Type – reflects the type of the object, read-only, cannot be associated with a variable.

Name – specifies the name of the object, cannot be associated with a variable.

Visible – determines whether the object is visible.

Color – specifies the color.

XYZs – specifies the starting polygon of the beam as a three column matrix whose rows represent the points (x-, y-, and z-coordinates) of the polygon.

dX, **dY**, **dZ** – specifies the x-, y-, and z-offset of the ending polygon.

Closed – specifies whether the beam ends are capped.

Transparency – specifies the object transparency, must be between 0 (fully opaque) and 1 (fully transparent).

4.2.7.10 Quadrangle

This object has the following properties:

Type – reflects the type of the object, read-only, cannot be associated with a variable.

Name – specifies the name of the object, cannot be associated with a variable.

Visible – determines whether the object is visible.

Color – specifies the color.

x1, **y1**, **z1**, **x2**, **y2**, **z2**, **x3**, **y3**, **z3**, **x4**, **y4**, **z4** – specify the coordinates of the four points of the quadrangle.

Transparency – specifies the object transparency, must be between 0 (fully opaque) and 1 (fully transparent).

4.2.7.11 Arrow

This object has the following properties:

Type – reflects the type of the object, read-only, cannot be associated with a variable.

Name – specifies the name of the object, cannot be associated with a variable.

Visible – determines whether the object is visible.

Color – specifies the color.

x1, **y1**, **z1** – specify the coordinates of the starting point.

nX, **nY**, **nZ** – specify the arrow vector.

Length – specifies the arrow length.

Cylinder radius – specifies the arrow cylinder radius.

Head radius – specifies the radius of the arrowhead cone.

Head length – specifies the height of the arrowhead cone.

Slices – specifies the number of segments used to model the arrow.

Transparency – specifies the object transparency, must be between 0 (fully opaque) and 1 (fully transparent).

4.2.7.12 Surface

This object has the following properties:

Type – reflects the type of the object, read-only, cannot be associated with a variable.

Name – specifies the name of the object, cannot be associated with a variable.

Visible – determines whether the object is visible.

Color – specifies the color.

x1, **y1**, **z1** – specify the x-, y-, and z-offset of the surface from the coordinate origin.

XYZs – specifies the coordinates of the surface points. This must be a three column matrix whose rows represent the points.

Points in row – specifies the number of points in each row of the surface.

Transparency – specifies the object transparency, must be between 0 (fully opaque) and 1 (fully transparent).

4.2.7.13 Text Label

This object has the following properties:

Type – reflects the type of the object, read-only, cannot be associated with a variable.

Name – specifies the name of the object, cannot be associated with a variable.

Visible – determines whether the object is visible.

Color – specifies the color.

x1, **y1**, **z1** – specify the coordinates of the base point.

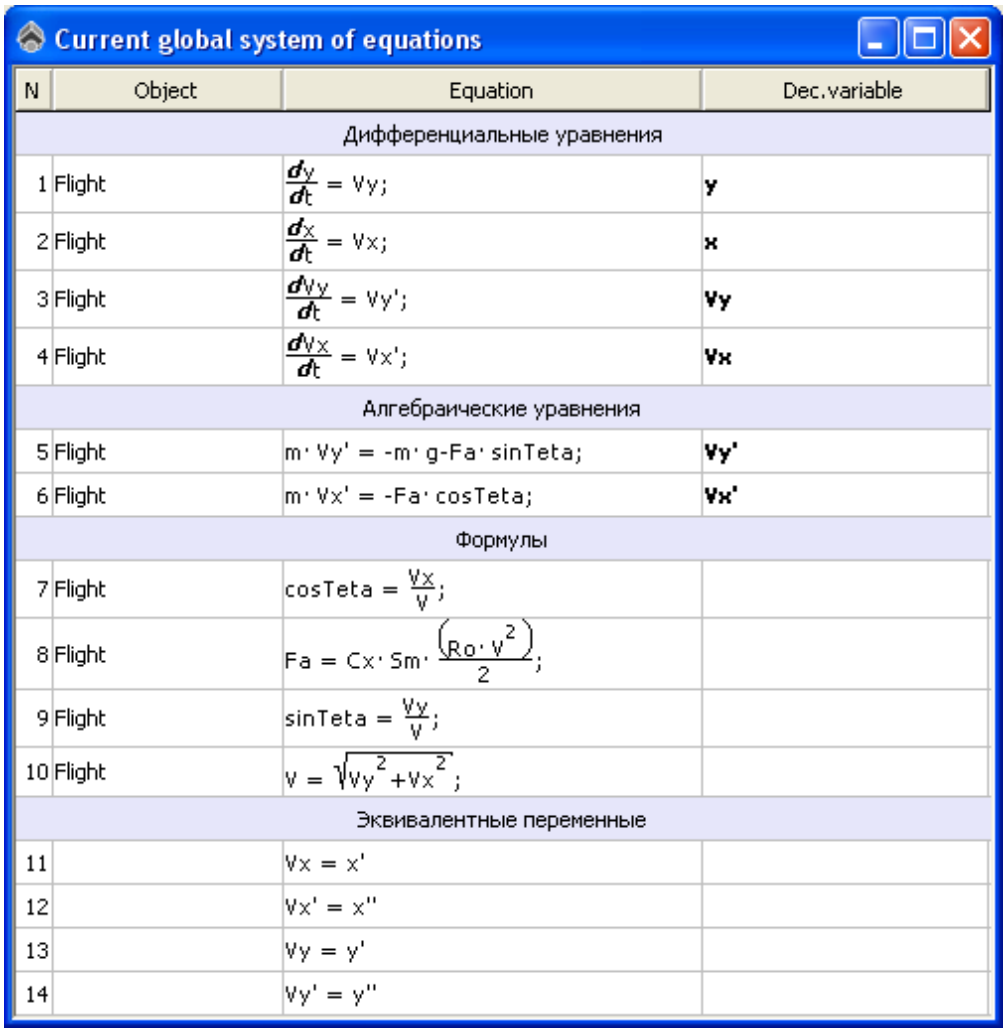
dX, **dY**, **dZ** – specify the offsets between the base point and the lower-left corner of the text.

Text – specify the label text.

Transparency – specifies the object transparency, must be between 0 (fully opaque) and 1 (fully transparent).

4.2.8 The “Aggregate System of Equations” Window

The **Aggregate System of Equations** window displays the current aggregate system of equations (see **Section 2.6**) of the model, as shown in **Figure 220**.



N	Object	Equation	Dec.variable
		Дифференциальные уравнения	
1	Flight	$\frac{dy}{dt} = Vy;$	**y**
2	Flight	$\frac{dx}{dt} = Vx;$	**x**
3	Flight	$\frac{dVy}{dt} = Vy';$	**Vy**
4	Flight	$\frac{dVx}{dt} = Vx';$	**Vx**
		Алгебраические уравнения	
5	Flight	$m \cdot Vy' = -m \cdot g - Fa \cdot \sin Teta;$	**Vy'**
6	Flight	$m \cdot Vx' = -Fa \cdot \cos Teta;$	**Vx'**
		Формулы	
7	Flight	$\cos Teta = \frac{Vx}{V};$	
8	Flight	$Fa = Cx \cdot Sm \cdot \frac{\left(Ro \cdot V^2\right)}{2};$	
9	Flight	$\sin Teta = \frac{Vy}{V};$	
10	Flight	$V = \sqrt{Vy^2 + Vx^2};$	
		Эквивалентные переменные	
11		$Vx = x'$	
12		$Vx' = x''$	
13		$Vy = y'$	
14		$Vy' = y''$	

Figure 220 The "Aggregate System of Equations" window

The window displays the equations in a table where each row represents an equation and has the following fields:

N – displays the equation number.

Object – displays the name of the object owning the equation.

Equation – displays the equation.

Dec.variable – displays the unknown variable associated with the equation, if any.

Equations are shown divided into the following categories:

Differential equations

Algebraic equations

Formulas

Equivalent variables

❖ **To open the "Aggregate System of Equations" window:**

1. Choose **Service | Show current equation system** from the main menu.

❖ **To toggle automatic column resizing:**

1. Right-click anywhere in the **Aggregate System of Equations** window and choose **Auto-resize columns** from the popup menu.

The contents of the **Aggregate System of Equations** window can at any time be saved as a text file or printed.

❖ **To save the contents of the "Aggregate System of Equations" window as a text file:**

1. Right-click anywhere in the **Aggregate System of Equations** window and choose **Save…** from the popup menu.
   The **Save As** dialog box will appear.
2. Browse to the folder where you want to save the file.
3. Type the file name in the **File name** field.
4. Click **Save**.

❖ **To print the contents of the "Aggregate System of Equations" window:**

1. Right-click anywhere in the **Aggregate System of Equations** window and choose **Print…** from the popup menu.
   The standard **Print** dialog box will appear.
2. Adjust printing properties, if necessary, and click **OK**.

4.2.9 The "Equation Blocks" Window

The **Equation Blocks** window displays information about equation blocks (see **Section 2.5**), as shown in **Figure 221**.

Figure 221 The "Equation Blocks" window

The information is organized in a table where each row represents a block and has the following fields:

Block – displays the name of the block.

Diff.equations – displays the number of differential equations in the block.

Alg.equations – displays the number of algebraic equations in the block.

Formulas – displays the number of formulas in the block.

In order for the **Equation Blocks** window to display information about equation blocks, the **Take equation blocks into account** option must be set in the numerical method settings (see **Section 4.2.13.2.6**).

❖ **To open the "Equation Blocks" window:**

1. Choose **Service | Show information about equation blocks** from the popup menu.

4.2.10 The "Calculator" Window

The **Calculator** window allows the user to evaluate expressions (see **Section 2.15**) involving model variables and parameters. The **Calculator** window looks as shown in **Section 2.15**.

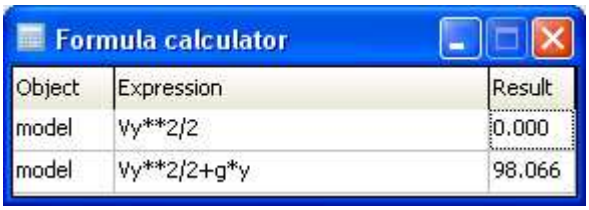


Figure 222 The "Calculator" window

❖ **To open the "Calculator" window:**

1. Click **Calculator** on the main toolbar, or
   Choose **Service | Calculator** from the main menu.
   The **Calculator** window will appear. If it has not been used before, it will look as shown in **Figure 223**.

Figure 223 An empty "Calculator" window

❖ **To enter an expression for evaluation:**

1. Choose a row in the **Calculator** window.
2. By default, the **Calculator** window evaluates entered expressions in the context of the model object. If you want to evaluate an expression in the context of some other object, specify the object in the **Object** field.
   To do so, either drag the object from a **Structure** or **Behavior Chart** window and drop it onto the **Object** field of the selected row, or right-click the **Object** field, choose **Change current object…** from the popup menu, and choose the object using the **Choose object** dialog box that will appear as shown in **Figure 224**.

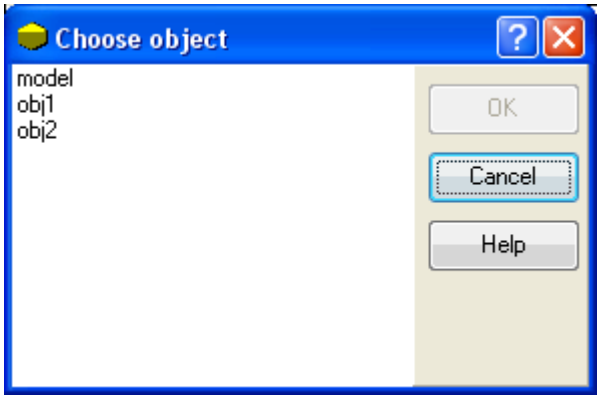


Figure 224 The "Choose object" dialog box

3. Click the **Expression** field of the selected row. After a short while, click the **Expression** field again.
   An in-place editing field will appear.
4. Type the expression.
5. Hit *Enter* or click anywhere outside the editing field. The value of the expression will appear in the **Result** field as shown in **Figure 225**.

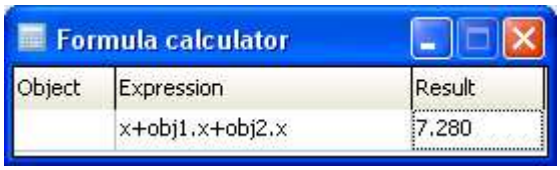


Figure 225 Evaluation result

❖ **To evaluate an already entered expression:**

1. Right-click the corresponding row and choose **Calculate row** from the popup menu.

❖ **To evaluate all available expressions:**

1. Right-click anywhere in the **Calculator** window and choose **Calculate all** from the popup menu.

❖ **To add a row:**

1. Right-click anywhere in the **Calculator** window and choose **Add line** from the popup menu.

❖ **To remove a row:**

1. Right-click the row and choose **Delete** line from the popup menu.

4.2.11 Model Time

The model time consists of two intertwined components: *continuous time* and *discrete time*. The continuous model time represents the time elapsed since the start of the model, in model time units. The discrete model time represents the number of behavior chart transition executions since the start of the model. The current continuous and discrete model times are displayed on the main toolbar of the visual executable model, see **Section 4.2**.

The visual executable model can run either as fast as possible or at a certain ratio between the continuous model time and the real time. This is configured on the **Run** tab of the **Model settings** dialog box, see **Section 4.2.13.1.1**. Running the model as fast as possible allows to simulate the model at the highest possible speed, whereas running the model at a fixed model-time-to-real-time ratio allows to animate the model smoothly. By default, every newly created model is configured to run as fast as possible.

4.2.12 The “Breakpoint conditions” Dialog Box

The **Breakpoint conditions** dialog box allows the user to configure the model to stop when a certain condition is satisfied. The supported condition types allow to stop the model:

Every N units of continuous or discrete time

At a given continuous or discrete time moment

When a specific behavior chart state is entered

When a specific or any behavior chart transition is executed

When a given conditional expression (predicate) becomes true

❖ **To open the “Breakpoint conditions” dialog box:**

1. Click **Breakpoint conditions** on the main toolbar, or
   Choose **Service | Breakpoint conditions** from the main menu.
   The **Breakpoint conditions** dialog box will appear as shown in **Figure 226**.

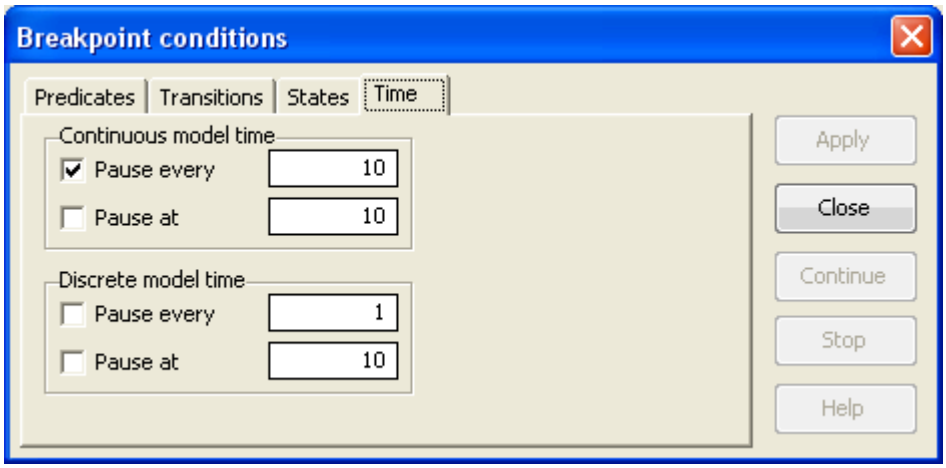


Figure 226 The "Breakpoint conditions" dialog box, the "Time" tab

The **Breakpoint conditions** dialog box consists of the following tabs: **Predicates**, **Transitions**, **States**, and **Time**. These tabs are described in the subsections that follow. The dialog box also contains the following buttons: **Apply**, **Close**, **Continue**, and **Stop**. The **Apply** button applies the changes made in the **Breakpoint conditions** dialog box. The **Close** button closes the **Breakpoint conditions** dialog box. The **Continue** button runs the model. The **Stop** button stops the model. For more information on model states and execution commands, see **Section 4.2.14**.

4.2.12.1 The "Time" Tab

The **Time** tab, see **Figure 226**, contains the following elements:

Continuous model time | Pause every – If this checkbox is checked, the model will stop every N continuous model time units, where N is specified in the field next to the checkbox.

Continuous model time | Pause at – If this checkbox is checked, the model will stop at the continuous model time moment specified in the field next to the checkbox.

Discrete model time | Pause every – If this checkbox is checked, the model will stop every N discrete model time units, where N is specified in the field next to the checkbox.

Discrete model time | Pause at – If this checkbox is checked, the model will stop at the discrete model time moment specified in the field next to the checkbox.

4.2.12.2 The "States" Tab

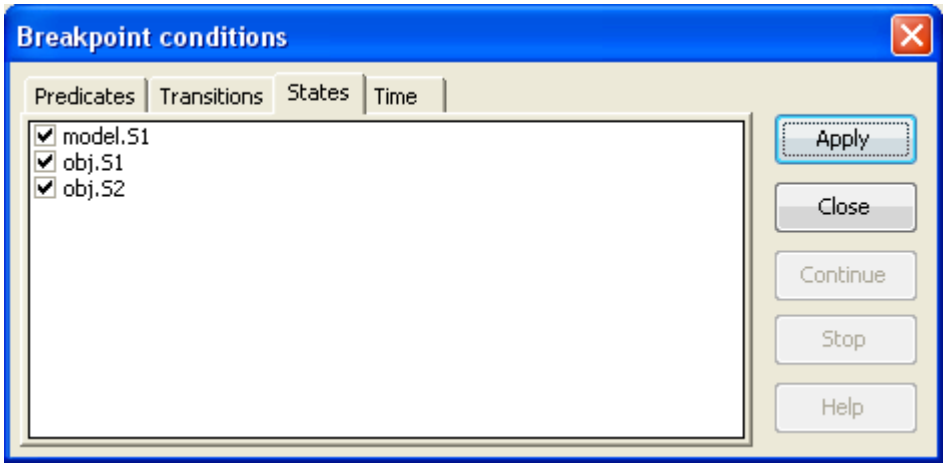


Figure 227 The "States" tab

The **States** tab, see **Figure 227**, allows the user to specify the states that, when entered, cause the model to stop. The tab shows the states in a list. Every state has a checkbox next to it. If a checkbox is checked, the model will stop every time the corresponding state is entered. The list can be populated by dragging and dropping states from **Behavior Chart** windows.

❖ **To add a state:**

1. Drag the state from the corresponding **Behavior Chart** window and drop it onto the list in the **States** tab.

❖ **To remove a state:**

1. Right-click the state in the list and choose **Delete…** from the popup menu. A confirmation message box will appear.
2. Click **Yes**.

4.2.12.3 The "Transitions" Tab

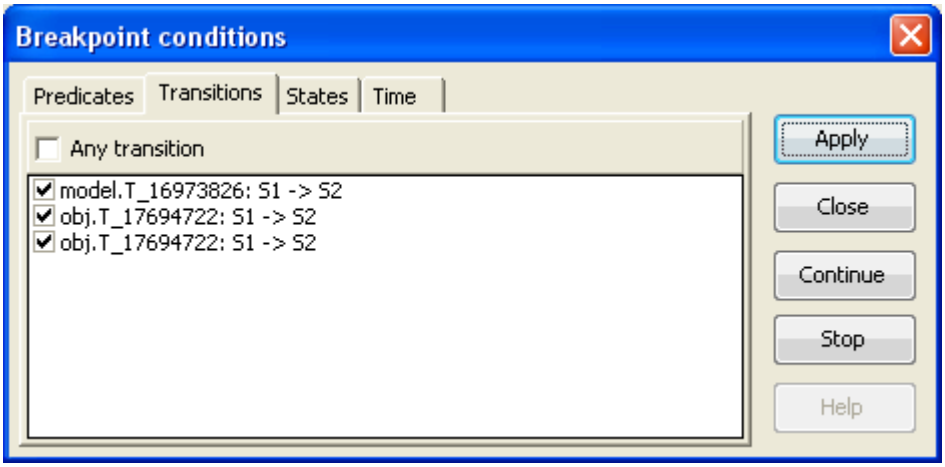


Figure 228 The "Transitions" tab

The **Transitions** tab, see **Figure 228**, allows the user to specify the transitions that, when executed, cause the model to stop. The tab shows the transitions in a list. Every transition has a checkbox next to it. If a checkbox is checked, the model will stop every time the corresponding transition is executed. The list can be populated by dragging and dropping transitions from **Behavior Chart** windows.

The additional **Any transition** checkbox, when checked, causes the model to stop on any transition in any behavior chart.

❖ **To add a transition:**

1. Drag the transition from the corresponding **Behavior Chart** window and drop it onto the list in the **Transitions** tab.

❖ **To remove a transition:**

1. Right-click the transition in the list and choose **Delete…** from the popup menu.
   A confirmation message box will appear.
2. Click **Yes**.

4.2.12.4 The “Predicates” Tab

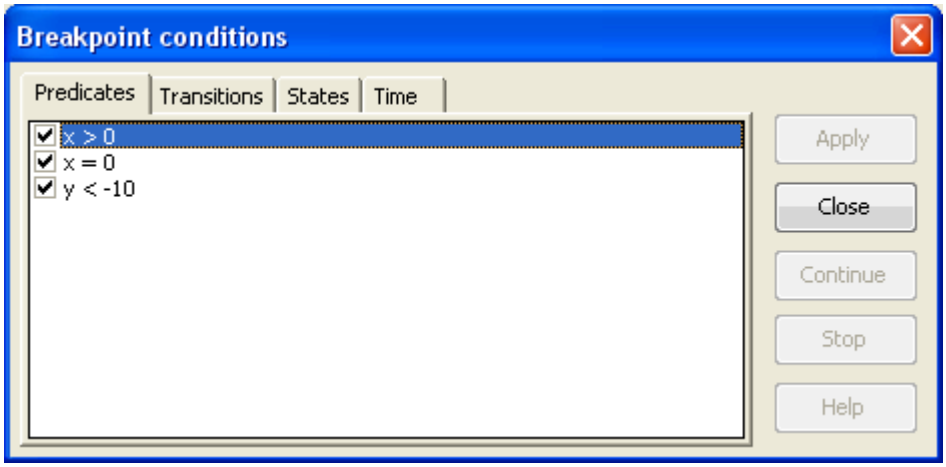


Figure 229The “Predicates” tab

The **Predicates** tab, see **Figure 229**, allows the user to specify one or more boolean conditions, also known as predicates, that, when satisfied, cause the model to stop. The tab shows the predicates in a list. Every predicate has a checkbox next to it. If a checkbox is checked, the model will stop every time the corresponding condition becomes true.

❖ **To add a predicate:**

1. Right-click anywhere in the list and choose **Add…** from the popup menu. The **Edit** predicate dialog box will appear as shown in **Figure 230**.

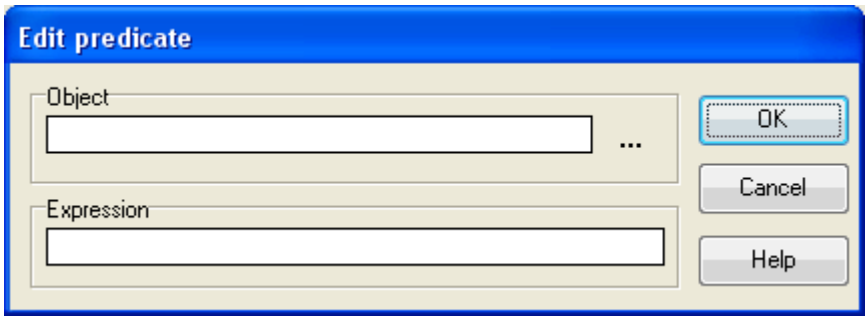


Figure 230 The “Edit predicate” dialog box

2. To select the object whose variables and parameters will participate in the predicate, click ... .
   The **Choose object** dialog box will appear as shown in **Figure 231**.

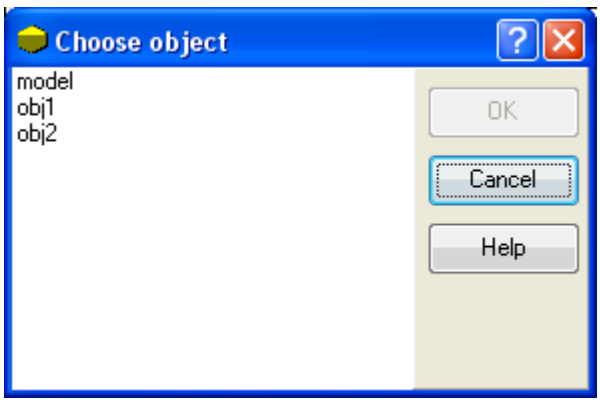


Figure 231 The “Choose object” dialog box

3. Select the object and click **OK**.
4. Type the predicate expression in the **Expression** field.
5. Click **OK**.

❖ **To modify a predicate:**

1. Right-click the predicate and choose **Edit…** from the popup menu.
   The **Edit predicate** dialog box will appear as shown in **Figure 230**.
2. Use the **Edit predicate** dialog box to modify the predicate.
3. Click **OK**.

❖ **To remove a predicate:**

1. Right-click the predicate and choose **Delete…** from the popup menu.
   A confirmation message box will appear.
2. Click **Yes**.

4.2.13 Model Settings

The visual model settings consist of the basic model settings and numerical method settings. The basic model settings are configured in the **Model settings** dialog box, see **Section 4.2.13.1**. The numerical method settings are configured in the **Numerical methods** dialog box, see **Section 4.2.13.2**.

At any time, the current visual model settings can be saved to or loaded from a file.

❖ **To save the current visual model settings to a file:**

1. Choose **Settings | Save to file…** from the main menu.
   The **Save settings** dialog box will appear as shown in **Figure 232**.

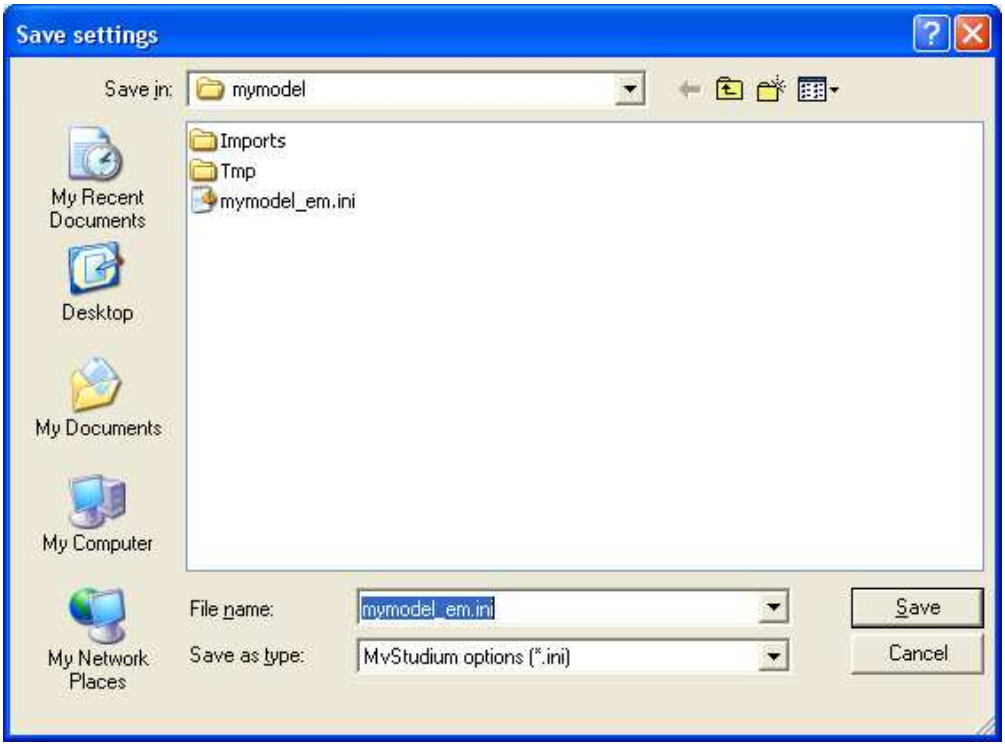


Figure 232 The "Save settings" dialog box

2. Browse to the folder where you want to save the current settings.

3. Type the file name in the **File name** field.
4. Click **Save**.

❖ **To load the visual model settings from a file:**

1. Choose **Settings | Load from file…** from the main menu.
   The **Load settings** dialog box will appear as shown in **Figure 233**.

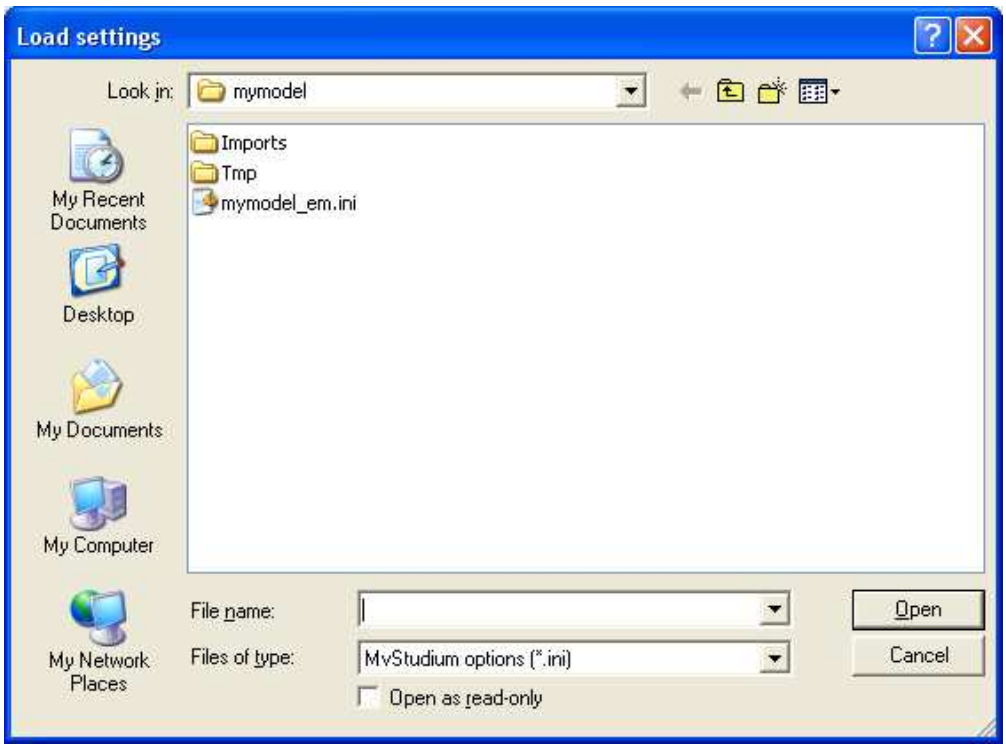


Figure 233 The "Load settings" dialog box

2. Select the file you want to open and click **Open**.

4.2.13.1 The "Model settings" dialog box

The **Model settings** dialog box allows the user to configure the basic model settings.

❖ **To open the "Model settings" dialog box:**

1. Click **Model settings** on the main toolbar, or
   Choose **Settings | Model…** from the main menu.
   The **Model settings** dialog box will appear as shown in **Figure 234**.

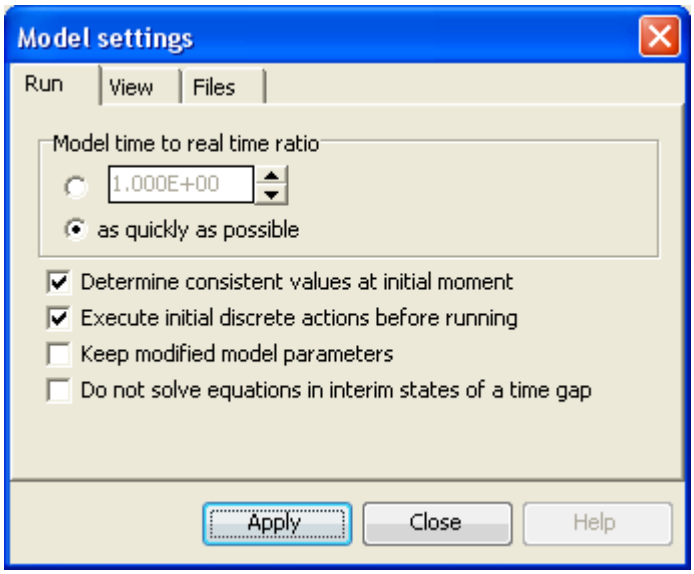


Figure 234 The "Model settings" dialog box, the "Run" tab

The **Model settings** dialog box consists of the following tabs: **Run**, **View**, and **Files**. These tabs are described in the subsections that follow. The dialog box also contains two buttons: **Apply** and **Close**. The **Apply** button applies the current settings and closes the **Model settings** dialog box. The **Close** button closes the dialog box without applying the current settings.

4.2.13.1.1 The "Run" Tab

The **Run** tab, see **Figure 234**, contains the following elements:

Model time to real time ratio – This section allows the user to configure the model execution speed. The **as quickly as possible** option instructs the model to run as fast as possible. The other option configures the model to run at a fixed model-time-to-real-time ratio, specified in the field next to the option. For more information, see **Section 4.2.11**.

Determine consistent values at initial moment – This checkbox determines whether the model attempts to find consistent variable values upon startup.

Execute initial discrete actions before running – This checkbox determines whether the model executes initial discrete actions upon startup.

Keep modified model parameters – This checkbox determines whether modified parameter values survive model restarts.

Do not solve equations in interim states of a time gap – If this checkbox is checked, the model does not attempt to analyze and solve the aggregate system of equations in interim states of time gaps.

4.2.13.1.2 The “View” Tab

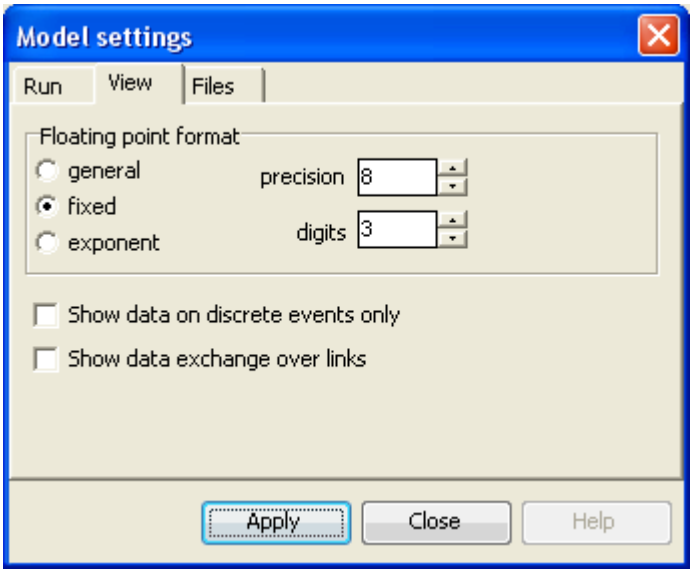


Figure 235 The “View” tab

The **View** tab, see **Figure 235**, contains the following elements:

Floating point format – This group of options and fields defines the format used to display floating-point numbers. The **general**, **fixed**, and **exponential** options select the general, fixed, and exponential floating-point formats respectively. The **precision** field specifies the maximum number of digits in the textual representation of floating-point numbers. The **digits** field specifies the number of decimal places.

Show data on discrete events only – If this checkbox is checked, **Variables** windows refresh their contents only when discrete events occur in the model.

Show data exchange over links – If this checkbox is checked, links are highlighted in **Structure** windows when the variables they connect change during model execution.

4.2.13.1.3 The “Files” Tab

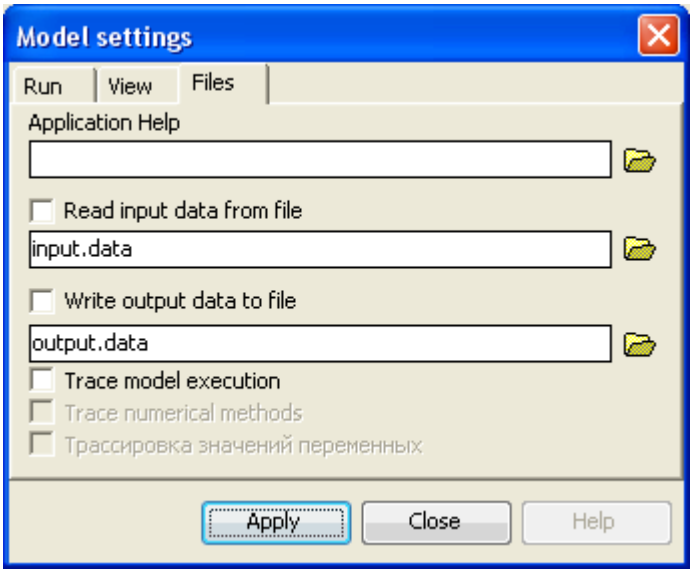


Figure 236 The “Files” tab

The **Files** tab, see **Figure 236**, contains the following elements:

Application Help – This field specifies the path to the help (`.hlp`) file associated with the model.

Read input data from file – If this checkbox is checked, the model, upon startup, loads parameter values and initial variable values from the text file specified in the next field, ignoring parameter values and initial variable values specified elsewhere.

Write output data to file – If this checkbox is checked, the model, upon termination, saves all parameter values and initial variable values to the text file specified in the next field.

Trace model execution – This checkbox enables/disables model execution tracing. When model execution tracing is enabled, the model logs detailed information about its execution to the `_trace.txt` file.

Trace numerical methods – This checkbox enables/disables tracing of numerical methods. This checkbox is only available if model execution tracing is enabled.

Trace variable values – This checkbox enables/disables tracing of variable values. This checkbox is only available if model execution tracing is enabled.

If the **Read input data from file** checkbox is checked, the text file specified after that checkbox must consist of lines of the following form:

```
<paramater_or_variable_name> = <value> // <comments>
```

If the **Write output data to file** checkbox is checked, the text file specified after that checkbox must consist of lines of the following form:

```
<paramater_or_variable_name> =
```

Upon termination, the model will modify each line of the output file by adding the value of the specified parameter or variable after the “=” sign.

4.2.13.2 The “Numerical methods” Dialog Box

The **Numerical methods** dialog box allows the user to choose and configure the numerical methods used to solve the aggregate system of equations of the model (see **Section 2.6**).

Rand Model Designer offers the following categories of numerical methods:

Automatic solvers. These methods analyze the aggregate system of equations and automatically select the method that suits best. If the selected method does not perform well, the detailed information about encountered problems is presented to the user. By analyzing that information, the user can better understand the characteristics of the system and gather enough knowledge to manually choose the right method.

Concrete methods. These methods attempt to solve the aggregate system of equations using some well known algorithm. These methods have well known characteristics that determine their suitability for solving different kinds of equation systems. If the concrete method selected by the user cannot solve the current aggregate system of equations with the requested tolerance, it stops the model execution and displays an error message.

Debug methods. These methods are designed to robustly solve the aggregate system of equations for the maximum possible period of model time, allowing the user to quickly analyze and, if necessary, correct the model. Debug methods are available for differential equations only.

❖ **To open the "Numerical methods" dialog box:**

1. Click **Numerical methods** on the main toolbar, or
   Choose **Settings | Numerical methods...** from the popup menu.
   The **Numerical methods** dialog box will appear as shown in **Figure 237**.

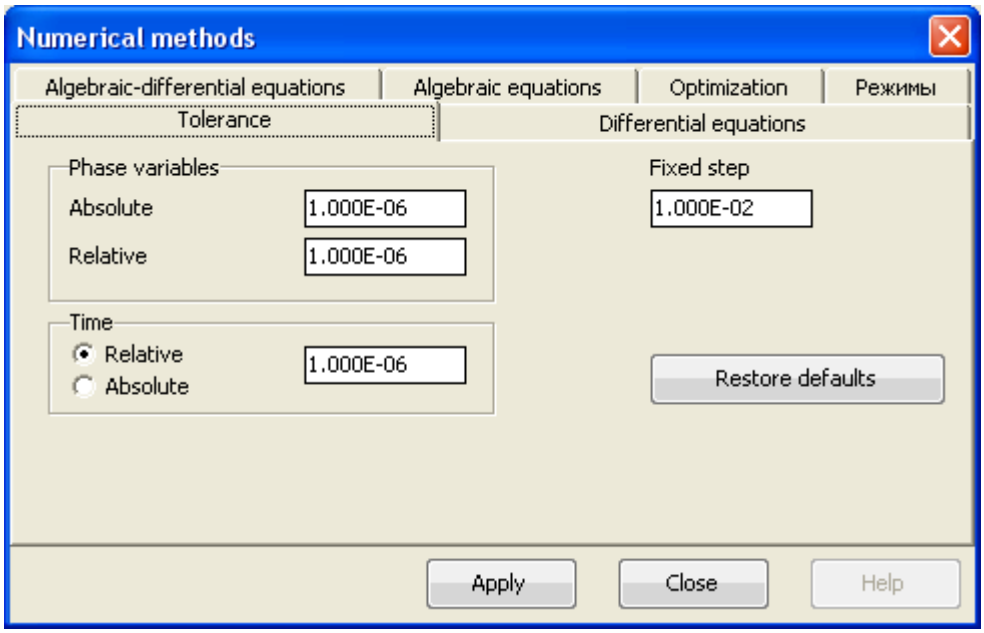


Figure 237 The "Numerical methods" dialog box, the "Tolerance" tab

The **Numerical methods** dialog box consists of the following tabs: **Tolerance**, **Differential equations**, **Algebraic-differential equations**, **Algebraic equations**, **Optimization**, and **Options**. These tabs are described in the subsections that follow. The dialog box also contains two buttons: **Apply** and **Close**. The **Apply** button applies the current settings and closes the dialog box. The **Close** button closes the dialog box without applying the current settings.

4.2.13.2.1 The "Tolerance" Tab

The **Tolerance** tab, see **Figure 237**, contains the following elements:

Phase variables – This group of fields specifies the absolute and relative error tolerances for variable values for all numerical methods and switch point detection.

Absolute – This field specifies the absolute error tolerance.

Relative – This field specifies the relative error tolerance.

Time – This group of elements specifies the time error tolerance for switch point detection.

Relative – This option selects the relative error tolerance. The tolerance value is specified in the field on the right.

Absolute – This option selects the absolute error tolerance. The tolerance value is specified in the field on the right.

Fixed step – This field specifies the fixed step for fixed-step numerical methods.

Restore defaults – This button restores the default tolerance settings.

4.2.13.2.2 The “Differential equations” Tab

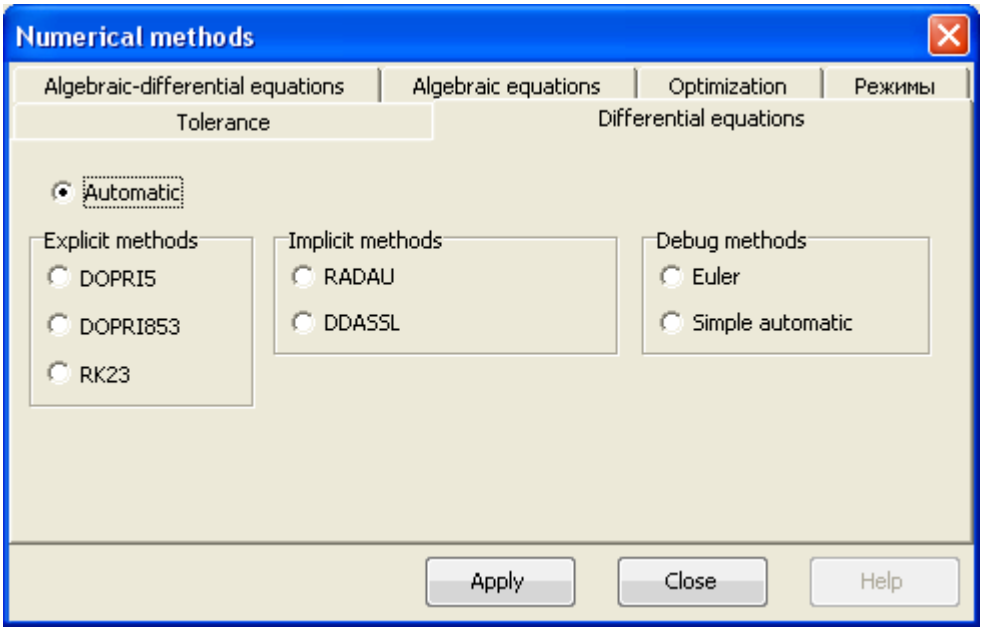


Figure 238 The “Differential equations” tab

The **Differential equations** tab, **Figure 238**, contains the following elements:

Automatic – This option selects the automatic solver.

Explicit methods – This group of options allows the user to select one of the following explicit methods:

DOPRI5

DOPRI853

RK23

Implicit methods – This group of options allows the user to select one of the following implicit methods:

RADAU

DDASSL

Debug methods – This group of options allows the user to select one of the following debug methods:

Euler

Simple automatic

4.2.13.2.3 The “Algebraic-differential equations” Tab

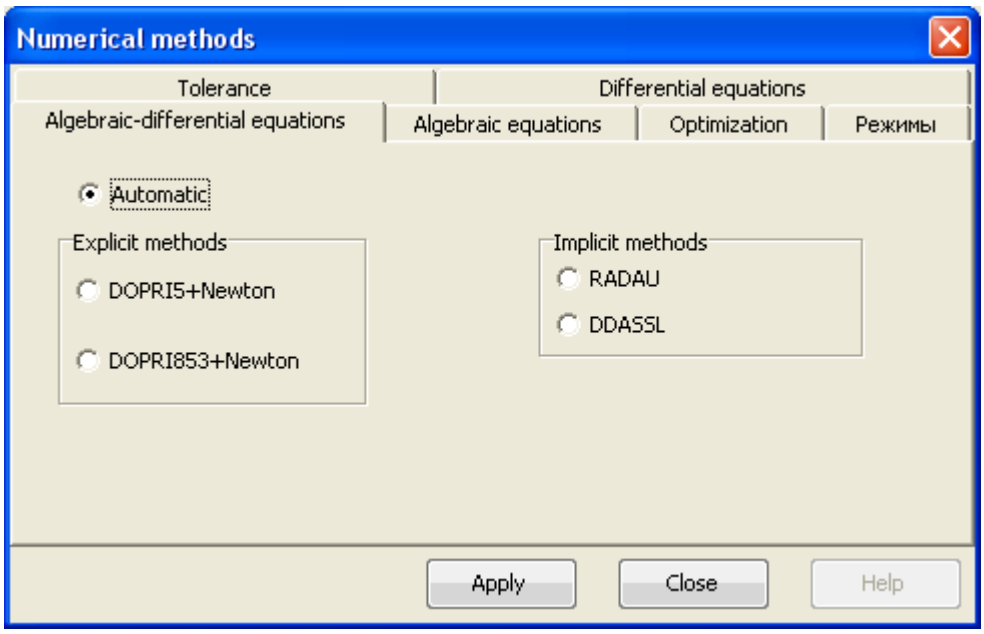


Figure 239 The “Algebraic-differential equations” tab

The **Algebraic-differential equations** tab, see **Figure 239**, contains the following elements:

Automatic – This option selects the automatic solver.

Explicit methods – This group of options allows the user to select one of the following explicit methods:

DOPRI5+Newton

DOPRI853+Newton

Implicit methods – This group of options allows the user to select one of the following implicit methods:

RADAU

DDASSL

4.2.13.2.4 The “Algebraic equations” Tab

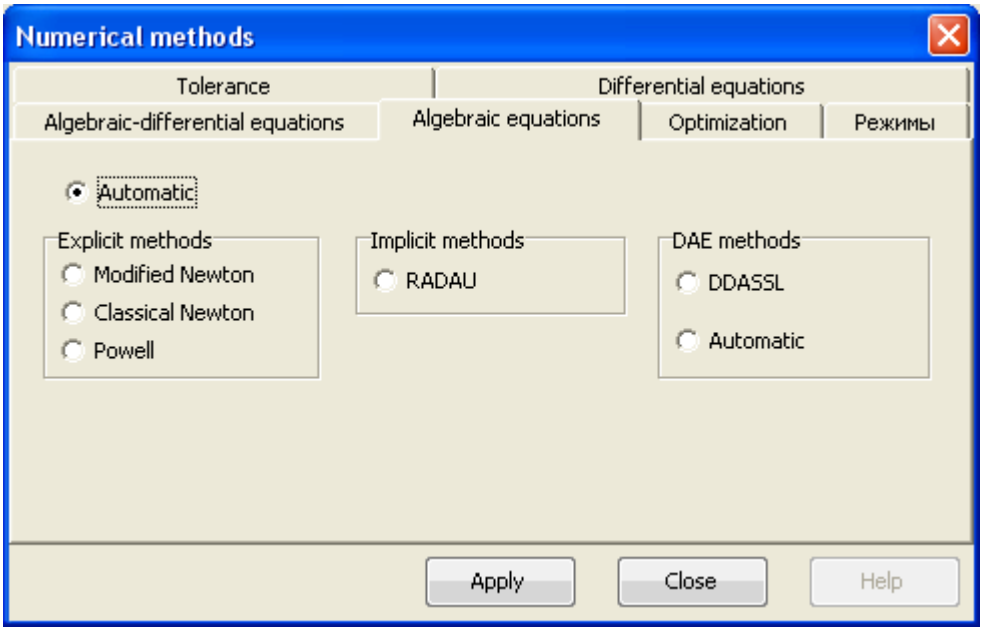


Figure 240 The “Algebraic equations” tab

The **Algebraic equations** tab, see **Figure 240**, contains the following elements:

Automatic – This option selects the automatic solver.

Explicit methods – This group of options allows the user to select one of the following explicit methods:

Modified Newton

Classical Newton

Powell

Implicit methods – This group of options allows the user to select the following implicit method:

RADAU

DAE methods – This group of options allows the user to select one of the following methods intended primarily for solving algebraic-differential equations:

DDASSL

Automatic

4.2.13.2.5 The “Optimization” Tab

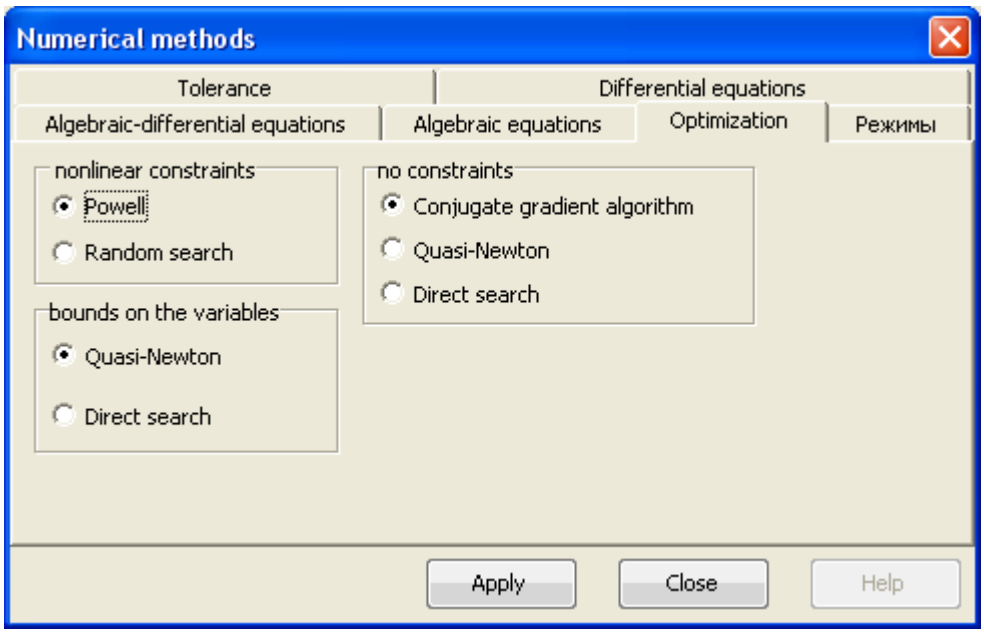


Figure 241 The “Optimization” tab

The **Optimization** tab, **Figure 241**, contains the following elements:

nonlinear constraints – This group of options specifies the method used to solve optimization problems with nonlinear constraints. Possible options:

Powell

Random-search

bounds on the variables – This group of options specifies the method used to solve optimization problems where variables have individual feasibility ranges. Possible options:

Quasi-Newton

Direct search

no constraints – This group of options specifies the method used to solve optimization problems without constraints. Possible options:

Conjugate gradient algorithm

Quasi-Newtown

Direct search

4.2.13.2.6 The “Options” Tab

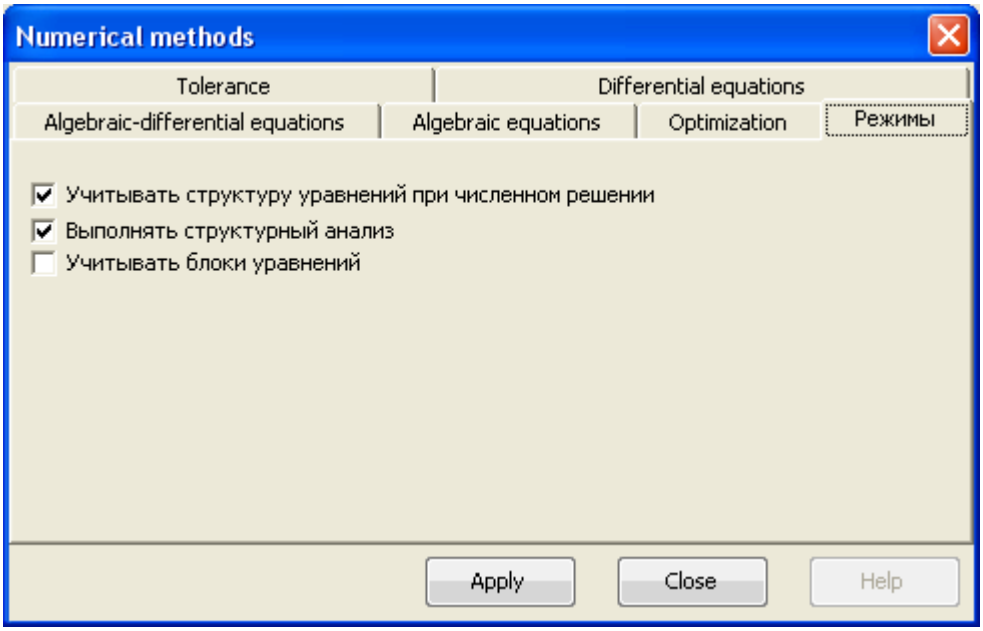


Figure 242 The “Options” tab

The **Options** tab, see **Figure 242**, contains the following elements:

Take system structure into account – If this checkbox is checked, numerical methods are instructed to take into account the structure of the aggregate system of equations.

Perform structure analysis – If this checkbox is checked, numerical methods are instructed to perform structure analysis of the aggregate system of equations.

Take equation blocks into account – If this checkbox is checked, numerical methods are instructed to take the declared equation blocks (see **Section 2.5**) into account.

These options are not available in Rand Model Designer Trial and Rand Model Designer Individual.

4.2.14 Executing the Model

At any time, the executable model can be in either running or stopped state. When the model is in the running state, its continuous and discrete times advance as the aggregate system of equations is solved and the behavior charts of hybrid objects are executed. When the model is in the stopped state, the model time does not change and no internal activity takes place within the model.

When the model is in the running state, it can at any time be stopped. When the model is in the stopped state, it can be either run, instructed to execute one discrete step, or restarted from the beginning.

❖ **To run the model:**

1. Click **Start** on the main toolbar, or
   Choose **Simulation | Start** from the main menu.

❖ **To stop the model:**

1. Click **Stop** on the main toolbar, or
   Choose **Simulation | Stop** from the main menu.

❖ **To execute one discrete step:**

1. Click **Discrete step** on the main toolbar, or
   Choose **Simulation | Discrete step** from the main menu.

❖ **To restart the model:**

1. Click **Restart** on the main toolbar, or
   Choose **Simulation | Restart** from the main menu.

4.2.15 The “Simulation Plan” Dialog Box

The **Simulation Plan** dialog box allows the user to create and activate the simulation plan. The simulation plan is a sequence of commands that, if active, is executed in parallel with the model. An example of the **Simulation Plan** dialog box containing a simple plan is shown in **Figure 243**.

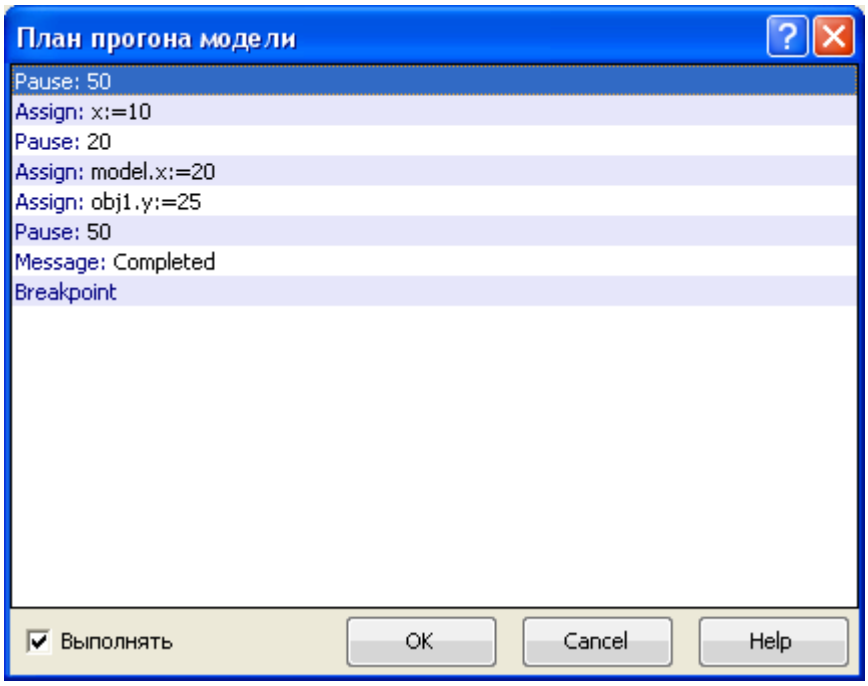


Figure 243 The “Simulation Plan” dialog box containing a simple plan

❖ **To open the “Simulation Plan” dialog box:**

1. Choose **Service | Plan** from the main menu, or

   If the simulation plan is already active, click **Simulation plan** on the main toolbar.
   The **Simulation Plan** dialog box will appear. If the simulation plan has not been configured yet, the dialog box will look as shown in **Figure 244**.

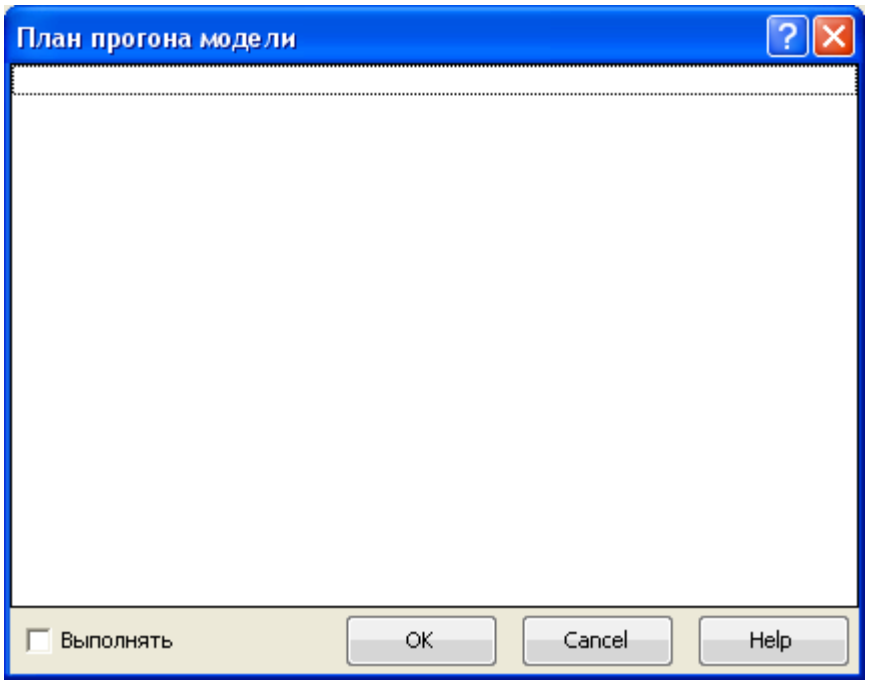


Figure 244 Empty simulation plan

The **Simulation Plan** dialog box displays the simulation plan in a list, one command per row. New commands can be either added to the end of the plan or inserted between existing commands. Existing commands can be edited or deleted. The **Active** checkbox in the lower left corner determines whether the plan is active or not. The **OK** button applies the current changes and closes the dialog box. The **Cancel** button closes the dialog box without applying the current changes.

The simulation plan can contain the following commands:

Assign – assigns a new value to a model variable.

Pause – delays the plan execution for a given amount of model time.

Message – displays a message.

Breakpoint – stops the model execution.

Quit – terminates the model.

Assignment commands are added to the plan by dragging respective variables from **Variables** and **Structure** windows. Other commands are added via the context menu.

❖ **To add or insert an assignment command:**

1. Drag the variable you want to assign from a **Variables** or **Structure** window and drop it onto the command list in the **Simulation Plan** dialog box. If you want to add the assignment to the end of the plan, drop the variable onto the empty area below the last command. Otherwise, if you want to insert the assignment before a specific command, drop it onto that command.
   The **Assignment** dialog box will appear as shown in **Figure 245**.

Figure 245 The "Assignment" dialog box

2. In the **Assignment** dialog box, type the value that the command should assign to the variable and click **OK**.

❖ **To add or insert a pause command:**

1. If you want to add a pause command to the end of the plan, right click anywhere in the command list and choose **Add | Pause…** from the popup menu. Otherwise, if you want to insert a pause command before a specific command, right-click that command and choose **Insert | Pause…** from the popup menu.
   The **Pause** dialog box will appear as shown in **Figure 246**.

Figure 246 The "Pause" dialog box

2. Type the amount of model time the command should pause for and click **OK**.

❖ **To add or insert a message command:**

1. If you want to add a message command to the end of the plan, right click anywhere in the command list and choose **Add | Message…** from the popup menu. Otherwise, if you want to insert a message command before a specific command, right-click that command and choose **Insert | Message…** from the popup menu.
   The **Message** dialog box will appear as shown in **Figure 247**.

Figure 247 The "Message" dialog box

2. Type the message text and click **OK**.

❖ **To add or insert a breakpoint command:**

1. If you want to add a breakpoint command to the end of the plan, right click anywhere in the command list and choose **Add | Breakpoint** from the popup menu. Otherwise, if you want to insert a breakpoint command before a specific command, right-click that command and choose **Insert | Breakpoint** from the popup menu.

❖ **To add or insert a quit command:**

1. If you want to add a quit command to the end of the plan, right click anywhere in the command list and choose **Add | Quit** from the popup menu. Otherwise, if you want to insert a quit command before a specific command, right-click that command and choose **Insert | Quit** from the popup menu.

❖ **To modify an existing assignment, pause, or message command:**

1. Double-click the command, or
   Right-click the command and choose **Edit…** from the popup menu.
   Depending on the command type, either the **Assignment**, **Pause**, or **Message** dialog box will appear.

2. Modify the command argument and click **OK**.

❖ **To delete a command:**

1. Right-click the command and choose **Delete…** from the popup menu.
   A confirmation message box will appear.

2. Click **Yes**.

4.2.16 Saving and Restoring the Current Model State

The current state of the visual executable model can at any time be saved to or restored from a file. By convention, model state files are given the extension `.mst`.

❖ **To save the current model state to a file:**

1. Choose **Simulation | Save model state…** from the main menu.
   The **Save model state** dialog box will appear as shown in **Figure 248**.

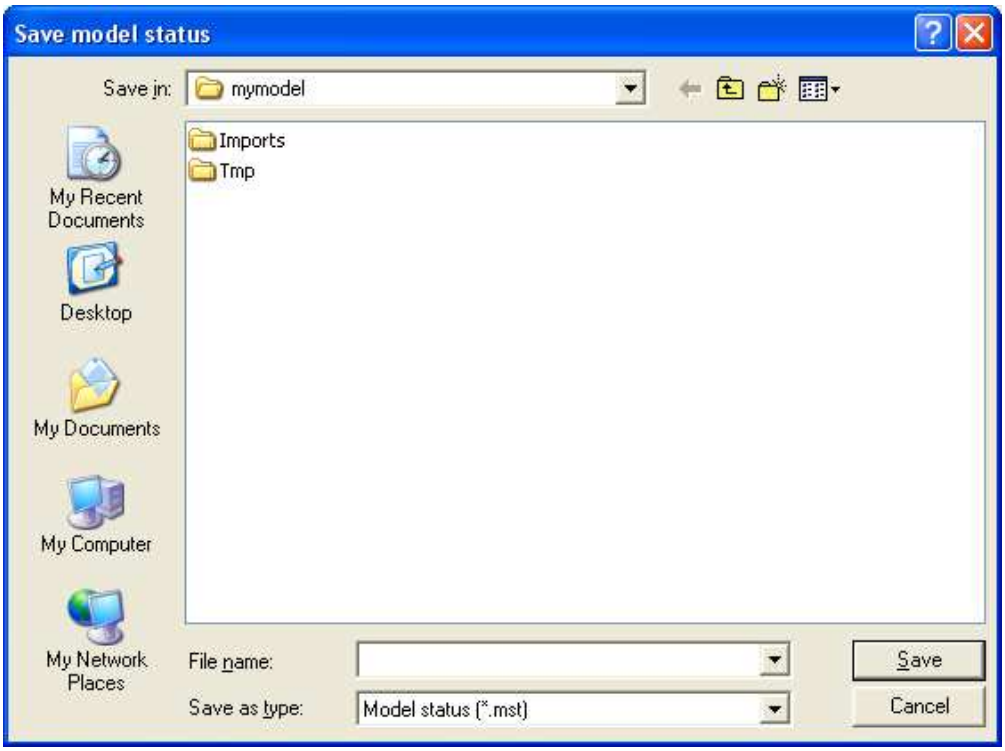


Figure 248 The "Save model state" dialog box

2. Browse to the folder where you want to save the file.

3. Type the file name in the **File name** field.

4. Click **Save**.

❖ **To restore the current model state from a file:**

1. Choose **Simulation | Load model state…** from the main menu.
   The **Load model state** dialog box will appear as shown in **Figure 249**.

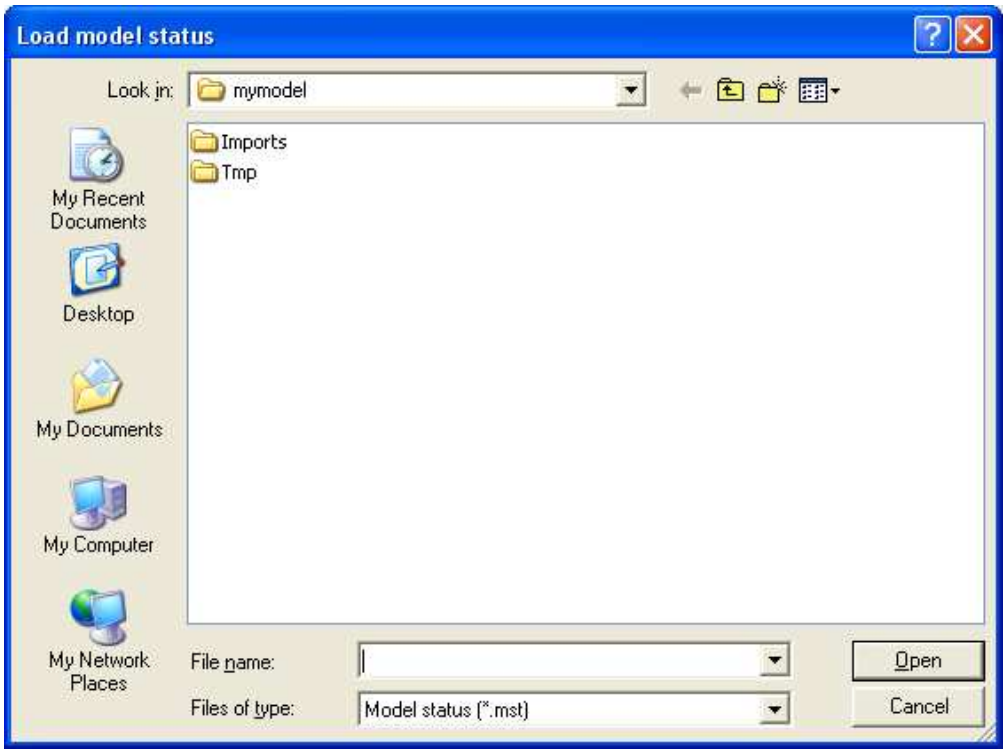


Figure 249 The "Load model state" dialog box

2. Select the file you want to restore the model state from and click **Open**.

4.2.17 Miscellaneous Functions

In addition to all the functionality described in this chapter, the visual executable model allows the user to:

Cascade, tile, and minimize open windows

Open the Rand Model Designer help

Open the visual executable model help

Terminate the model

❖ **To cascade open windows:**

1. Choose **Window | Cascade** from the main menu.

❖ **To tile open windows:**

1. Choose **Window | Tile** from the main menu.

❖ **To minimize all open windows:**

1. Choose **Window | Minimize all** from the main menu.

❖ **To open the Rand Model Designer help:**

1. Choose **Help | Help…** from the main menu.

❖ **To open the visual executable model help:**

1. Choose **Help | Application help…** from the main menu.

❖ **To terminate the visual executable model:**

1. Choose **Settings | Exit** from the main menu.

2. If the model contains unsaved changes, a confirmation message box will appear.
   If you want to save the changes, click **Yes**. Otherwise, click **No**.

4.2.18 Using the Standalone Visual Executable Model

Rand Model Designer Corporate and Rand Model Designer Educational allow the user to generate the visual executable model as a standalone executable that can be run separately from the integrated modeling environment.

❖ **To generate the standalone visual executable model:**

1. Make sure the **Visual model (Win32 Application)** option is selected on the **Model** tab of the **Project** options dialog box (see **Section 3.12.3**).
2. Build the standalone executable model.
   To do so, click **Build model** on the main toolbar, or
   Choose **Model | Build…** from the popup menu.
   Rand Model Designer will build the model and display a success message box.
   Click **OK** in the message box.
3. Save the generated model with the desired name to the desired location.
   To do so, choose **Model | Save as…** from the main menu.
   The **Save model as** dialog box will appear.
   In the **Save model as** dialog box, specify the name and location and click **Save**.

Rand Model Designer will copy the executable model to the specified location along with the following files:

`_mathmvs.dll` – the numerical methods library

User-supplied DLLs implementing external functions and procedures, if any

`<model_name>_vm.ini` – the default model settings file

The model-specific help file, if it is specified in the model settings

❖ **To run the standalone executable model:**

1. Run the generated executable.

Every standalone visual executable model supports the following command line options:

`-A` (or `/A`) – Causes the model to run automatically upon startup.

`-S <path_to_settings_file>` (or `/S <path_to_settings_file>`) – Instructs the model to load the settings from the specified file.

The standalone visual executable model loads its settings and window layout from the default settings file upon startup and saves them to the default settings file upon termination, unless the `-S` command line option is specified. If the `-S` command line option is specified, the supplied settings file is used instead of the default one.

4.3 EMBEDDABLE EXECUTABLE MODEL

The embeddable executable model is a Windows DLL that any application can link to in order to get programmatic access to the model.

❖ **To generate the embeddable executable model:**

1. Make sure the **Embedded model (Win32 DLL)** option is selected on the **Model** tab of the **Project** options dialog box (see **Section 3.12.3**).

2. Build the model.
   To do so, click **Build model** on the main toolbar, or
   Choose **Model | Build…** from the popup menu.
   Rand Model Designer will build the model and display a success message box.
   Click **OK** in the message box.

3. Save the generated model with the desired name to the desired location.
   To do so, choose **Model | Save as…** from the main menu.
   The **Save model as** dialog box will appear.
   In the **Save model as** dialog box, specify the name and location and click **Save**.

The embeddable model exports a set of functions that allow the application to control model execution and access the contents of the model. These functions are described in detail in the subsections that follow.

The embeddable model allows the application to run multiple model instances simultaneously. Every model instance maintains its own model time and state of objects and is independent from other model instances. Every model instance is identified by a unique handle of type `TModelHandle`. Many of the functions of the embeddable model take a `TModelHandle` as the first parameter, allowing the caller to identify the model instance. Some functions do not take an explicit `TModelHandle` parameter: these work with the implicit default model instance whose `TModelHandle` is assumed to be `0`. The default model instance is created automatically upon the first call to a function that requires the default instance. Other model instances need to be created explicitly using the `createModel` function.

The embeddable model uses the numerical method settings that were active when the visual executable model was executed last time before the embeddable model was built. These settings can be altered using the functions exported by the embeddable model.

4.3.1 Data Types and Constants

The embeddable model functions make use of the following data types and constants:

Data types:

`integer` – a 32-bit signed integer.

`pChar` – a pointer to a null-terminated string of 8-bit characters.

`pWideChar` – a pointer to a null-terminated string of 16-bit characters.

`double` – a 64-bit floating-point ANSI/IEEE Std 754-1985 number.

`pointer` – a 32-bit pointer to data.

`variant` – a value of undefined type.

`THandle=integer` – a handle.

`TModelHandle=THandle` – a handle to a model instance.

`TVarHandle=THandle` – a handle to a variable.

`TRetCode=integer` – a function return code.

`TRetCode` constants:

- `rcOK=0` – success.
- `rcModelDoesNotExist=1` – the specified model instance does not exist.
- `rcModelIsNotAtBreakpoint=2` – the specified model instance is not in the stopped state.
- `rcModelError=3` – model error detected.
- `rcInvalidExpression=4` – invalid expression.
- `rcTimeOut=5` – the operation timed out.
- `rcInvalidModelHadle=6` – the specified model handle is invalid.
- `rcInvalidVarHandle=7` – the specified variable handle is invalid.
- `rcVarDoesNotHaveValue=8` – the specified variable does not have a value at this point.
- `rcInvalidVarType=9` – invalid variable type.
- `rcInsufficientDataLength=10` – buffer size is too small.
- `rcIllegalVarValue=11` – illegal variable value.
- `rcVarNotFound=12` – the specified variable was not found.
- `rcExceptionWhileExecuting=-1` – the operation threw an exception.

`TVarHandle` constants:

- `vhVarNotFound=-1` – the specified variable was not found.

4.3.2 Model Creation and Destruction Functions

`function createModel(var hModel:TModelHandle):TRetCode;` – Creates a new model instance. On success, returns a handle to the created instance in the `hModel` parameter. The returned handle is always greater than or equal to 0. The embeddable model will never assign the returned handle to any other model instance. When a created model instance is no longer needed, it should be destroyed using the `destroyModel` function.

`function destroyModel(hModel:TModelHandle):TRetCode;` – Destroys the model instance specified by the `hModel` parameter.

4.3.3 Functions to Save and Restore Model State

`function saveModel(hModel:TModelHandle; var size:integer; pData:pointer):TRetCode;` – Saves the current state of the specified model instance to the specified buffer in XML format. The `hModel` parameter identifies the model instance. The `pData` parameter points to the buffer. The `size` parameter specifies the actual buffer size on entry and the minimum buffer size required to store all available data on exit. This function is typically used as follows: The function is first called with `size` set to `0` and `pData` set to `nil` to obtain the required buffer size. A buffer of the required size is than allocated and the function is called again to retrieve all available data. The function fills the supplied buffer with an XML document that can be viewed and edited by the user.

`function loadModel(hModel:TModelHandle; size:integer; pData:pointer):TRetCode;` – Loads the state of the specified model instance from the specified buffer. The `hModel` parameter identifies the model instance. The `pData` parameter points to the buffer. The `size` parameter specifies the buffer size.

4.3.4 Model Execution Functions

`function runToEx(hModel:TModelHandle; Tau:double): TRetCode;` – Runs the specified model instance for the specified amount of continuous model time. The `hModel` parameter identifies the model instance. The `Tau` parameter specifies the amount of model time in model time units. The function returns only after all necessary computations are completed and all variables are updated accordingly. Upon return from this function, the model instance is guaranteed to be in the stopped state.

`function RunTo(Tau:double):TRetCode;` – Runs the default model instance for the specified amount of continuous model time. A call to this function is equivalent to `runToEx(0, Tau)`.

`function RunThreadTo(Tau:double; var hThread:THandle): TRetCode;` – Runs the default model instance for the specified amount of continuous model time in a separate thread. Creates a separate thread that calls `RunTo(Tau)`, and returns immediately. The `hThread` parameter receives a handle to the created thread on success or 0 otherwise. The caller can detect the moment when the model stops by calling the `Stopped` function periodically or handling the message configured using the `SetEndMessage` function.

`function runUntilEx(hModel:TModelHandle; C:pChar): TRetCode;` – Runs the specified model instance until the specified conditional expression evaluates to true. The `hModel` parameter identifies the model instance. The `C` parameter specifies the conditional expression. The function returns only after all necessary computations are completed and all variables are updated accordingly. Upon return from this function, the model instance is guaranteed to be in the stopped state. The model has to interpret the C condition, which may negatively affect the model performance. The most efficient types of conditions that require no interpretation are:

Signal names

`Finalized`

Expressions of the form `"time = <value>"`

`function RunUntil(C:pChar):TRetCode;` – Runs the default model instance until the specified conditional expression evaluates to true. A call to this function is equivalent to `runUntilEx(0, C)`.

`function runUntilStopEx(hModel:TModelHandle; TimeOut:integer):TRetCode;` – Runs the specified model instance until all discrete actions of the model are completed or the specified timeout expires, whatever happens first. The `hModel` parameter identifies the model instance. The `TimeOut` parameter specifies the timeout in milliseconds. The function returns only after all necessary computations are completed and all variables are updated accordingly. Upon return from this function, the model instance is guaranteed to be in the stopped state. The function returns `rcTimeOut` if the timeout expires or `rcOK` otherwise.

`function RunUntilStop(TimeOut:integer):TRetCode;` – Runs the default model instance until all discrete actions of the model are completed or the specified timeout expires, whatever happens first. A call to this function is equivalent to `runUntilStopEx(0, TimeOut)`.

`function RunThreadUntil(C:pChar):TRetCode;` – Runs the default model instance in a separate thread until the specified conditional expression evaluates to true. Creates a separate thread that calls `RunUntil(C)`, and returns immediately. The caller can detect the moment when the model stops by calling the `Stopped` function periodically or handling the message configured using the `SetEndMessage` function.

`function Restart:TRetCode;` – Restarts the default model instance, that is, resets the model time to 0 and brings all model objects to their initial states. The function stops the model if it is currently running.

`function Stop(TimeOut:integer):TRetCode;` – Stops the default model instance at the nearest possible model time moment. The `TimeOut` parameter specifies the wait timeout in milliseconds.

`function Stopped:TRetCode;` – Returns `rcOK` if the model is in the stopped state or `rcModelIsNotAtBreakpoint` otherwise.

All functions except for `Stop`, `Stopped`, and `Restart` may only be called when the target model instance is in the stopped state.

4.3.5 Variable Management Functions

`function getVarHandleEx(hModel:TModelHandle; VarName:pWideChar; var hVar:TVarHandle):TRetCode;` – Looks up a variable by name in the specified model instance. The `hModel` parameter identifies the model instance. The `VarName` parameter specifies the variable name. The `hVar` parameter receives a handle to the variable with the specified name if it exists, or `vhVarNotFound` otherwise.

`function GetVarHandle(VarName:pChar):TVarHandle;` – Looks up a variable by name in the default model instance. The `VarName` parameter specifies the variable name. The function returns a handle to the variable with the specified name if it exists, or `vhVarNotFound` otherwise.

`function getVarTypeEx(hModel:TModelHandle; hVar:TVarHandle; var VarType:integer):TRetCode;` – Retrieves the type of the specified variable. The `hModel` parameter identifies the model instance. The `hVar` parameter identifies the variable. On success, the function stores the type code in the `VarType` parameter and returns `rcOK`. Otherwise, `VarType` is set to 0 and an appropriate error code is returned. The possible type codes are the same as with the `GetVariables` function.

`function getVarValueEx(hModel:TModelHandle; hVar:TVarHandle; var Y:variant):TRetCode;` – Retrieves the current value of the specified variable of arbitrary type. The `hModel` parameter identifies the model instance. The `hVar` parameter identifies the variable. The `Y` parameter receives the current value of the specified variable.

`function GetVarValue(hVar:TVarHandle; var Y:variant):TRetCode;` – Retrieves the current value of the specified variable of the default model instance. A call to this function is equivalent to `getVarValueEx(0, hVar, Y)`.

`function getVarDoubleValueEx(hModel:TModelHandle; hVar:TVarHandle; var Y:double):TRetCode;` – Retrieves the current value of the specified `double` variable. The `hModel` parameter identifies the model instance. The `hVar` parameter identifies the variable. The `Y` parameter receives the current value of the specified variable.

`function GetVarDoubleValue(hVar:TVarHandle; var Y:double):TRetCode;` – Retrieves the current value of the specified `double` variable of the default model instance. A call to this function is equivalent to `getVarDoubleValueEx(0, hVar, Y)`.

`function getVarVectorValueEx(hModel:TModelHandle; hVar:TVarHandle; var N:integer; var Y:TArrayDouble):TRetCode;` – Retrieves the current value of the specified variable of type `vector[N]`. The `hModel` parameter identifies the model instance. The `hVar` parameter identifies the variable. The `N` parameter specifies the vector size. The `Y` parameter receives the current value of the specified variable.

`function GetVarVectorValue(hVar:TVarHandle; var N:integer; var Y:TArrayDouble):TRetCode;` – Retrieves the current value of the specified variable of type `vector[N]` in the default model instance. A call to this function is equivalent to `getVarVectorValueEx(0, hVar, N, Y)`.

`function setVarValueEx(hModel:TModelHandle; hVar:TVarHandle; const X:variant):TRetCode;` – Assigns the specified value to the specified variable of arbitrary type. The `hModel` parameter identifies the model instance. The `hVar` parameter identifies the variable. The `X` parameter specifies the value. If the type of `X` does not match the variable type, the function attempts to convert `X` to the variable type. If that fails, `rcIllegalVarValue` is returned. Assigning `TRUE` to a signal variable sends a signal without parameters to that variable.

`function SetVarValue(hVar:TVarHandle; X:variant):TRetCode;` – Assigns the specified value to the specified variable of arbitrary type in the default model instance. A call to this method is equivalent to `setVarValueEx(0, hVar, X)`.

`function setVarDoubleValueEx(hModel:TModelHandle; hVar:TVarHandle; X:double):TRetCode;` – Assigns the specified value to the specified variable of type `double`. The `hModel` parameter identifies the model instance. The `hVar` parameter identifies the variable. The `X` parameter specifies the value.

`function SetVarDoubleValue(hVar:TVarHandle; X:double): TRetCode;` – Assigns the specified value to the specified variable of type `double` in the default model instance. A call to this method is equivalent to `setVarDoubleValueEx(0, hVar, X)`.

`function setVarVectorValueEx(hModel:TModelHandle; hVar:TVarHandle; var N:integer; var X:TArrayDouble):TRetCode;` – Assigns the specified value to the specified variable of type `vector[N]`. The `hModel` parameter identifies the model instance. The `hVar` parameter identifies the variable. The `N` parameter specifies the vector size. The `X` parameter specifies the value.

`function SetVarVectorValue(hVar:TVarHandle; var N:integer; var X:TArrayDouble):TRetCode;` – Assigns the specified value to the specified variable of type `vector[N]` in the default model instance. A call to this method is equivalent to `setVarVectorValueEx(0, hVar, N, X)`.

`function getTimeValueEx(hModel:TModelHandle; var Time:double):TRetCode;` – Retrieves the current continuous model time of the specified model instance. The `hModel` parameter identifies the model instance. The `Time` parameter receives the current continuous model time. If the specified model instance does not exist or is not in the stopped state, `Time` is set to `NaN` and an appropriate error code is returned.

`function GetTimeValue:double;` – Returns the current continuous model time of the default model instance, or `NaN` if the default model instance is not in the stopped state.

4.3.6 Numerical Methods Configuration Functions

`procedure setTol(ATOL,RTOL,TTOL:double);` – Sets the error tolerances. The `ATOL` parameter specifies the absolute error tolerance for variable values. The `RTOL` parameter specifies the relative error tolerance for variable values. The `TTOL` parameter specifies the absolute time tolerance for switch point detection. This function affects all model instances.

`procedure setNumMethods(ODE,DAE,NAE:integer);` – Selects the methods to be used to solve the aggregate system of equations. The `ODE` parameter specifies the method that should be used in case the aggregate system of equations consists solely of differential equations. The `DAE` parameter specifies the method that should be used in case the aggregate system of equations contains both differential and algebraic equations. The `NAE` parameter specifies the method that should be used in case the aggregate system of equations consists solely of algebraic equations. The selection affects all model instances.

Possible values of the `ODE` parameter:

`nmAutoODE=13` – automatic

`nmDOPRI5=2` – DOPRI5

`nmDOPRI853=4` – DOPRI853

`nmODEX=3` – ODEX

`nmDDASSL=8` – DDASSL

`nmRADAU5=10` – RADAU5

`nmEiler=18` – Euler

`nmRK4=1` – RK4

Possible values of the `DAE` parameter:

`nmAutoDAE=14` – automatic

`nmDOPRI5_N =20` – DOPRI5+Newton

`nmDOPRI853_N =21` – DOPRI853+Newton

`nmODEX_N =22` – ODEX+Newton

Possible values of the `NAE` parameter:

`nmAutoNAE=12` – automatic

`nmNewtonF=15` – “quick” Newton

`nmNewtonM=11` – “modified” Newton

`nmNewtonC=6` – “classical” Newton

`nmPowell=7` – Powell

4.3.7 Miscellaneous Functions

`function getLastHModErrorEx(hModel:TModelHandle; var str:variant):TRetCode;` – Retrieves the textual description of the last error. The `hModel` parameter identifies the model instance. The `str` parameter receives the description. If any of the embeddable model functions returns something other than `rcOK`, this function can be called to obtain the textual description of the error.

`function GetLastHModError:pChar;` – Returns the textual description of the last error occurred in the default model instance. If any of the embeddable model functions returns something other than `rcOK` for the default model instance, this function can be called to obtain the textual description of the error.

`function getMessages(hModel:TModelHandle; var Messages:variant):TRetCode;` – Retrieves the messages and warnings generated by the specified model instance since model start or the last call to this function. The `hModel` parameter identifies the model instance. The `Messages` parameter receives the list of messages and warnings.

`function getExpressionValueEx(hModel:TModelHandle; pC:pWideChar; var Value:variant):TRetCode;` – Evaluates the specified expression. The `hModel` parameter identifies the model instance. The `pC` string specifies the expression. The `Value` parameter receives the evaluation result. If the specified expression is incorrect, the `Value` parameter is set to `Unassigned`.

`function GetExpressionValue(pC:pChar):variant;` – Evaluates the specified expression in the context of the default model instance. Returns the evaluation result. The `pC` string specifies the expression. If the expression is incorrect, `Unassigned` is returned.

`function GetVariables(var vars:variant):TRetCode;` – Retrieves information about all variables of the default model instance. The `vars` parameter receives the result in the form of a two-dimensional `[1..N, 1..3]` variant array where `N` is the total number of variables in the model. Every `[i,1]`-th element of the array contains a string specifying the full name of the `i`-th variable. Every `[i,2]`-th element contains an integer code specifying the type of the `i`-th variable. Every `[i,3]`-th element contains `TRUE` if the `i`-th variable is independent or `FALSE` otherwise. Possible type codes are:

`double` – 4

`integer` – 5

`byte` – 6

`short` – 7

`boolean` – 9

`enumerated_type` – 10

`char` – 11

`string` – 12

`vector` – 14

`matrix` – 15

`signal` – 17

`function GetVariablesEx(var vars:variant):TRetCode;` –Retrieves information about the model object of the default model instance. The `vars` parameter receives the result in the form of a variant array with three elements. The first element is a string specifying the object identifier. The second element is an array describing the variables of the object in the same format as the result of the `GetVariables` function except for that the first element of each variable description specifies the variable identifier instead of the full name. The third element is an array describing the local objects of the model object.

`procedure SetEndMessage(hW,hM:integer);` – Specifies the message to be sent by threads created by the `RunThreadTo` function to inform the caller that the model has stopped. The `hW` parameter specifies the window handle. The `hM` parameter specifies the message code. If `hW` is non-zero, every thread created by the `RunThreadTo` function will call `PostMessage(hW, hM, 0, 0)` after the model reaches the stopped state.

CHAPTER 5
SysLib, The Standard Class Library

This chapter describes the classes that comprise the SysLib library

5.1 LINEAR BLOCKS

5.1.1 CGain

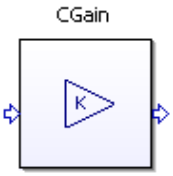


Figure 250 CGain

Class name: `CGain`

Parameters:

```
K: double := 1; -- the gain factor
X0: double := 0; -- the initial value of X
```

Input variables:

```
X: double := X0;
```

Output variables:

```
Y: double := 0;
```

Function:

This block generates its output by multiplying its input by the specified gain factor, implementing the function $Y = K \cdot X$.

5.1.2 CSummator

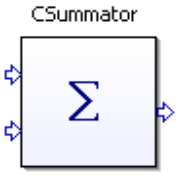


Figure 251 CSummator

Class name: `CSummator`

Parameters:

```
X0: double := 0; -- the initial value of X
Y0: double := 0; -- the initial value of Y
```

Input variables:

```
X: double := X0;
Y: double := Y0;
```

Output variables:

```
Z: double := 0;
```

Function:

This block outputs the sum of the inputs, implementing the function $Z = X + Y$.

5.1.3 CSubtractor

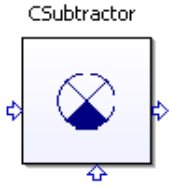


Figure 252 CSubtractor

Class name: `CSubtractor`

Parameters:

```
X0: double := 0; -- the initial value of X
Y0: double := 0; -- the initial value of Y
```

Input variables:

```
X: double := X0;
Y: double := Y0;
```

Output variables:

```
Z: double := 0;
```

Function:

This block implements the function $Z = X - Y$.

5.1.4 CIntegrator

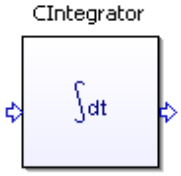


Figure 253 CIntegrator

Class name: `CIntegrator`

Parameters:

```
K: double := 1; -- the gain factor
X0: double := 0; -- the initial value of X
Y0: double := 0; -- the initial value of Y
```

Input variables:

```
X: double := X0;
```

Output variables:

```
Y: double := Y0;
```

Function:

This block integrates its input and outputs the result, the integral. The block implements the transfer function $W(s) = \frac{K}{s}$, which is equivalent to the equation $\frac{dY}{dt} = K \cdot X$.

5.1.5 CDifferentiator

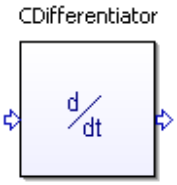


Figure 254 CDifferentiator

Class name: `CDifferetiator`

Parameters:

```
X0: double := 0; -- the initial value of X
Y0: double := 0; -- the initial value of Y
```

Input variables:

```
X: double := X0;
```

Output variables:

```
Y: double := Y0;
```

Function:

This block implements the equation $\frac{dX}{dt} = Y$.

5.1.6 CAperiodicUnit

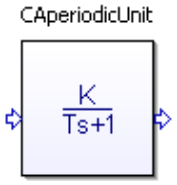


Figure 255 CAperiodicUnit

Class name: `CAperiodicUnit`

Parameters:

K: double := 1; -- the gain factor

T: double := 1; -- the time constant

X0: double := 0; -- the initial value of X

Y0: double := 0; -- the initial value of Y

Input variables:

X: double := X0;

Output variables:

Y: double := Y0;

Function:

This block implements the transfer function $W(s) = \frac{K}{T \cdot s + 1}$, which is equivalent to the equation $\frac{dY}{dt} = \frac{K \cdot X - Y}{T}$.

5.1.7 COscillatingUnit

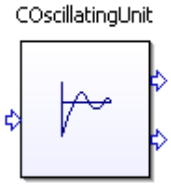


Figure 256 COscillatingUnit

Class name: `COscillatingUnit`

Parameters:

K: double := 1; -- the gain factor

T: double := 1; -- the time constant

Dzeta: double := 1; -- the damping factor

X0: double := 0; -- the initial value of X

Y0: double := 0; -- the initial value of Y

Input variables:

X: double := X0;

Output variables:

Y: double := Y0;

Vy: double := 0;

Function:

This block implements the transfer function $W(s) = \frac{K}{T^2 \cdot s^2 + 2 \cdot T \cdot \xi \cdot s + 1}$, which is equivalent to the following system of equations:

$$\begin{cases} \frac{dY}{dt} = Vy \\ \frac{dVy}{dt} = \frac{K \cdot X - 2 \cdot \xi \cdot T \cdot Vy - Y}{T^2} \end{cases}$$

5.2 NONLINEAR BLOCKS

5.2.1 CDeadZone

Figure 257 CDeadZone

Class name: `CDeadZone`

Parameters:

```
K: double := 1; -- the gain factor
D: double := 1; -- the dead zone width
X0: double := 0; -- the initial value of X
```

Input variables:

```
X: double := X0;
```

Output variables:

```
Y: double := 0;
```

Function:

This block implements the function shown in **Figure 258**.

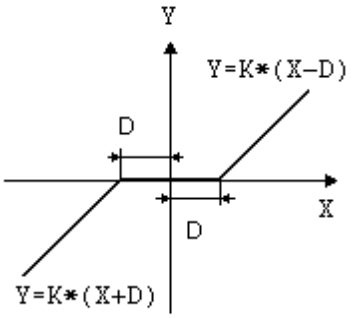


Figure 258 CDeadZone function

5.2.2 CSaturation

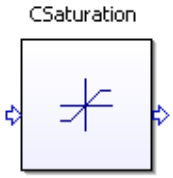


Figure 259 CSaturation

Class name: `CSaturation`

Parameters:

K: double := 1; -- the gain factor

UpperLimit: double := 1; -- the upper bound on the input signal

LowerLimit: double := - UpperLimit; -- the lower bound on the input signal

X0: double := 0; -- the initial value of X

Input variables:

X: double := X0;

Output variables:

Y: double := 0;

Function:

This block implements the function shown in **Figure 260**.

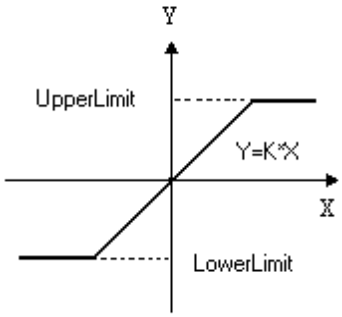


Figure 260 CSaturation function

5.2.3 CRelay

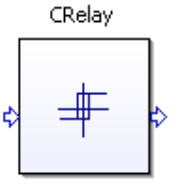


Figure 261 CRelay

Class name: `CRalay`

Parameters:

```
OnValue: double := 1; -- the output when the relay is on
OffValue: double := 0; -- the output when the relay is off
OnInput: double := 0; -- the on threshold
OffInput: double := 0; -- the off threshold
```

Input variables:

```
X: double := 0;
```

Output variables:

```
Y: double:= OffValue;
```

Function:

This block implements the function shown in **Figure 262**.

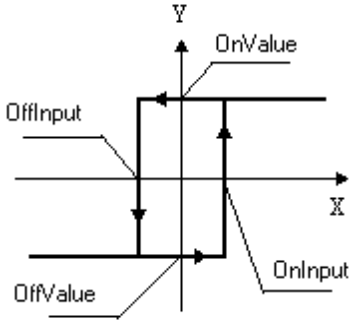


Figure 262 CRelay function

5.2.4 CSwitch

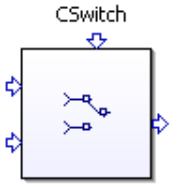


Figure 263 CSwitch

Class name: `CSwitch`

Parameters:

initX1: double := 0; -- the initial value of X1

initX2: double := 0; -- the initial value of X2

initZ: boolean := false; -- the initial value of Z

Input variables:

X1: double := initX1; -- the first input

X2: double := initX2; -- the second input

Z: boolean := initZ; -- the control input

Output variables:

Y: double := initX1;

Function:

This block implements the function Y=if Z then X2 else X1.

5.2.5 CInterpolator

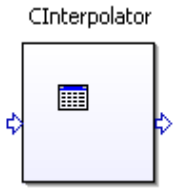


Figure 264 CInterpolator

Class name: `CInterpolator`

Parameters:

FX: vector := [0;1]; -- the vector of argument values

FY: vector := [0;1]; -- the vector of function values tabulated for FX

Input variables:

X: double := 0;

Output variables:

Y: double := 0;

Function:

This block generates its output by interpolating the function whose values FY are tabulated for points FX, using the block input as the function argument. The block raises an error if the input goes outside the interval spanned by FX.

Note:

The FX and FY vectors can be loaded from file(s) using the `VectorFromFile` function, for example:

```
static Int:
SysLib.CInterpolator(FX=VectorFromFile("fun1.txt","X"),
                     FY=VectorFromFile("fun1.txt"."Y"));
```

5.3 SIGNAL SOURCES (GENERATORS)

5.3.1 CRampGenerator

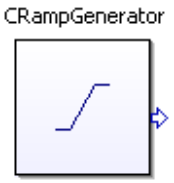


Figure 265 CRampGenerator

Class name: `CRampGenerator`

Parameters:

InitialOutput: double := 0; -- the initial value of the signal

UpperLimit: double := 100; -- the maximum value of the generated signal

Slope: double := 1; -- the rate of change of the generated signal

StartTime: double := 0; -- the time at which the signal begins to be generated

Input variables: none

Output variables:

Y: double:= InitialOutput;

Function:

This block implements the function shown in **Figure 266**.

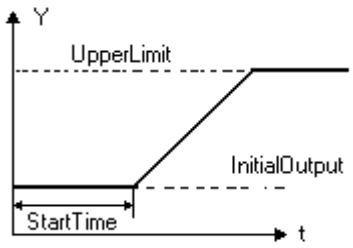


Figure 266 CRampGenerator function

5.3.2 CSineGenerator

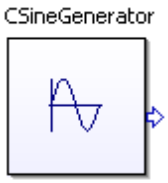


Figure 267 CSineGenerator

Class name: `CSineGenerator`

Parameters:

Amplitude: double := 1; -- the amplitude of the signal

Period: double := 2; -- the period of the signal

InitialPhase: double := 0; -- the phase shift, in radians

Input variables: none

Output variables:

```
Y: double:= Amplitude*sin(InitialPhase);
```

Function:

This block implements the function $Y = Amplitude \cdot \sin\left(\frac{2\pi \cdot t}{Period} + InitialPhase\right)$.

5.3.3 CSawtoothGenerator

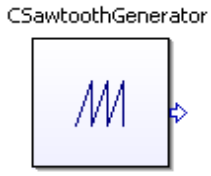


Figure 268 CSawtoothGenerator

Class name: `CSawtoothGenerator`

Parameters:

```
UpperLevel: double := 1; -- the upper bound
LowerLevel: double := 0; -- the lower bound
Period: double := 1; -- the period of the signal
```

Input variables: none

Output variables:

```
Y: double:= LowerLevel;
```

Function:

This block implements the function shown in **Figure 269**.

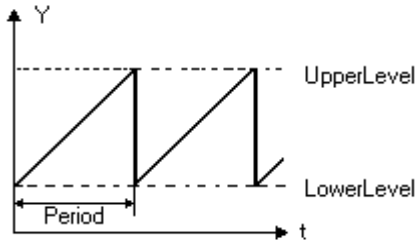


Figure 269 CSawtoothGenerator function

5.3.4 CPulseGenerator

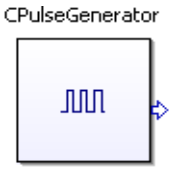


Figure 270 CPulseGenerator

Class name: `CPulseGenerator`

Parameters:

Amplitude: double := 1; -- the pulse amplitude

Period: double := 2; -- the pulse period

PulseWidth: double := Period/2; -- the pulse width, in model time units

Y0: double := 0; -- the average value of the output

Input variables: none

Output variables

Y: double:= Y0;

Function:

This block generates the output as shown in **Figure 271**.

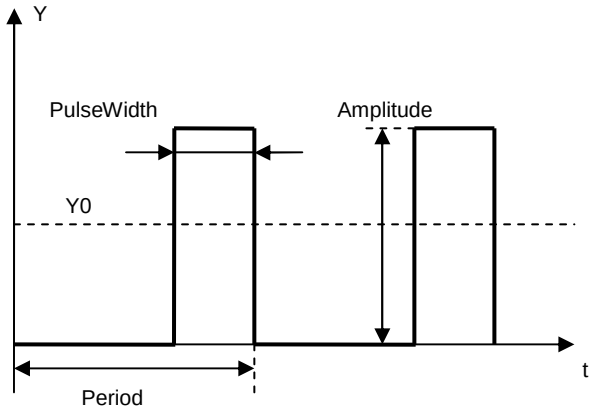


Figure 271 CPulseGenerator function

5.3.5 CStepsGenerator

Figure 272 CStepsGenerator

Class name: `CStepsGenerator`

Parameters:

Values: vector := [0;1]; -- the sequence of output values

Delays: vector := [1;2]; -- the sequence of time intervals

Input variables: none

Output variables:

Y: double:= Values [1];

Function:

This block outputs a periodic sequence of constant signal levels. The block keeps its output at `Values[1]` for the amount of time `Delays[1]`, then at `Values[2]` for the amount of time `Delays[2]`, and so on until the last element, and then restarts from the beginning.

5.3.6 CClock

Figure 273 CClock

Class name: `CClock`

Parameters: none

Input variables: none

Output variables:

T: double:= Time;

Function:

This block outputs the current model time, that is, implements the function `T=Time`.

CHAPTER 6
Appendices

This chapter contains appendices to the
User Manual

6.1 EXTENDED BACKUS-NAUR FORM

The Extended Backus-Naur Form (EBNF) is a formal notation used to describe the syntax of context-free grammars. The EBNF specifies a grammar as a finite set of production rules that define how all legal sentences of the grammar can be generated.

Production rules have the following form:

```
<non_terminal_symbol> ::= ebnf_expression
```

where `<non_terminal_symbol>` is a *non-terminal symbol* and `ebnf_expression` is an expression consisting of non-terminal symbols, *terminal symbols*, and special *operations*. Terminal symbols are fixed character strings without any grammatical structure. Non-terminal symbols are named elements that can consist of terminal and non-terminal symbols as defined by production rules. Non-terminal symbol names are by convention enclosed in angle brackets. Every production rule indicates that the non-terminal symbol on the left can be produced as specified by the expression on the right.

EBNF expressions support the following operations:

Concatenation. Terminal and non-terminal symbols are concatenated as they appear in the expression. For example, the rule `<A> ::= B<C>` indicates that `<A>` is a concatenation of `B` and `<C>`.

Choice. The exclamation mark between two or more symbols or sub-expressions indicates that the result can be generated by following any of the alternatives. For example, the rule `<A> ::= B!C!<D>` indicates that `<A>` is either `B`, `C`, or `<D>`.

Optional inclusion. The square brackets denote a sequence that may or may not be included in the generated sequence. For example, the rule `<A> ::= [B][<C>]` indicates that `<A>` is either `B`, `<C>`, a concatenation of `B` and `<C>`, or empty.

Repetition. The curly braces denote a sequence that may be repeated any number of times, including zero. For example, the rule `<A> ::= {B}` indicates that `<A>` is either empty or consists of one or more `B`s.

The following is an EBNF example specifying the syntax of natural numbers (non-negative integers):

```
<natural_number> ::= <digit>{<digit>}

<digit> ::= 0!1!2!3!4!5!6!7!8!9
```

6.2 IDENTIFIER

An *identifier* is a sequence of characters used to identify some entity in a project.

Identifiers obey the following rules:

Every identifier starts with a letter and contains only letters, digits, and underscores.

English and Russian Cyrillic letters are allowed and may be mixed within a single identifier.

Other characters are not allowed.

Reserved keywords (`begin`, `end`, etc) may not be used as identifiers.

Identifiers are case sensitive.

Identifiers may have arbitrary length.

6.3 THE MVL LANGUAGE

The MVL language is a format used to represent Rand Model Designer projects in textual form.

The MVL language has the following syntax:

```
<module> ::= <package_definition>

<package_definition> ::=
  package <project_name> is
    import <list_of_imported_packages>;
    {<project_element_definition>}
    <Model_class_definition>
    static model:Model
    [(<actual_initial_values>)];
  end <project_name>;

<project_name> ::= <name>

<name> ::=
  <identifier>!<package_name>.<identifier>

<package_name> ::= <identifier>

<list_of_imported_packages> ::=
  <package_name>{,<package_name>}

<project_element_definition> ::=
  <class_definition>
  !<global_constant_definition>
  !<global_function_or_procedure_definition>
  !<declared_type>

<class_definition> ::=
  [<class_stereotype>] class <class_name> is
    {<parameter_definition>}
    {<external_variable_definition>}
    {<variable_definition>}
    {<constant_definition>}
    {<function_or_procedure_definition>}
    [<structure_diagram_definition>]
    [<behavior_definition>]
  end <class_name>;

<class_stereotype> ::=
  [local][<visibility>]<behavior_type>

<visibility> ::= public!private

<behavior_type> ::= continuous!hybrid

<class_name> ::= <identifier>

<parameter_definition> ::=
  parameter <variable_definition>
```

```
<external_variable_definition> ::=
  <external_variable_stereotype><variable_definition>

<external_variable_stereotype> ::=
  input!output!flow!contact!connector

<variable_definition> ::=
  <variable_name>:<variable_type>
  [:=<initial_value>];

<variable_type> ::= <type>

<variable_name> ::= <identifier>

<initial_value> ::= <value>

<constant_definition> ::=
  constant <constant_name>:<constant_type>:=<value>;

<constant_type> ::= <type>

<constant_name> ::= <identifier>

<function_or_procedure_definition> ::=
  <function_or_procedure_specification> is
    {<local_variable_definition>}
    <function_or_procedure_body>

<function_or_procedure_specification> ::=
  <procedure_specification>!<function_specification>

<procedure_specification> ::=
  procedure <procedure_name>
  [<formal_parameters>]

<procedure_name> ::= <identifier>

<function_specification> ::=
  function <function_name>[<formal_parameters>]
    return <result_type>

<function_name> ::= <identifier>

<formal_parameters> ::=
  (<parameter_specification>{;<parameter_specification>})

<parameter_specification> ::=
  <parameter_category>
  <parameter_name>{,<parameter_name>}:
  <parameter_type>

<parameter_category> ::= [in]!out!in out

<parameter_name> ::= <identifier>

<parameter_type> ::= <type>

<result_type> ::= <type>

<local_variable_definition> ::=
  <variable_name>{,<variable_name>}:
  <variable_type>[:=<expression>];

<variable_name> ::= <identifier>
```

```
<function_or_procedure_body> ::=
  begin
    <operator_sequence>
  end <function_or_procedure_name>;

<function_or_procedure_name> ::=
  <function_name>!<procedure_name>

<operator_sequence> ::=
  {<operator>[<comments>]}

<operator> ::=
  <assignment_operator>
  !<procedure_call_operator>
  !<if_operator>
  !<case_operator>
  !<loop_operator>
  !<exit_operator>
  !<return_operator>
  !<send_operator>
  !<optimization_operator>

<comments> ::= <text_until_end_of_line>

<assignment_operator> ::= <identifier>:=<expression>;

<procedure_call_operator> ::=
  <procedure_name>[<argument_list>];

<argument_list> ::=
  (<argument>{,<argument>})

<argument> ::= <expression>

<if_operator> ::=
  if <conditional_expression> then
    <operator_sequence>
  {elsif <conditional_expression> then
    <operator_sequence>}
  [else
    <operator_sequence>]
  end if;

<conditional_expression> ::= <expression>

<case_operator> ::=
  case <selector_expression> of
  {when <option>{!<option>}=>
    <operator_sequence>}
  [when others =>
    <operator_sequence>]
  end case;

<option> ::= <static_expression>

<static_expression> ::= <expression>

<loop_operator> ::=
  [<loop_specification>]
  <loop_body>
```

```
<loop_specification> ::=
  for <loop_variable> in <range>
  !while <conditional_expression>

<loop_variable> ::= <identifier>

<range> ::= <lower_bound>..<upper_bound>

<lower_bound> ::= <static_expression>

<upper_bound> ::= <static_expression>

<loop_body> ::=
  loop
    <operator_sequence>
  end loop;

<exit_operator> ::=
  exit [when <conditional_expression>];

<return_operator> ::= return <expression>;

<send_operator> ::=
  send!^ [broadcast]
    <variable_name>[<argument_list>];

<optimization_operator> ::=
  <direction><double_expression> for
    <variable_specification>{,<variable_specification>}
    [constraints "{"<constraint>;{<constraint>;}"}"];

<direction> ::= minimize!maximize

<double_expression> ::= <expression>

<variable_specification> ::=
  <variable_name >[ in <range>]

<constraint> ::=
  <double_expression> <= <double_expression>
  !<double_expression> >= <double_expression>
  !<double_expression> = <double_expression>

<structure_diagram_definition> ::=
  {<object_definition>}{<link_definition>}

<object_definition> ::=
  static <object_name>:<class_name>
    [<actual_initial_values>];

<object_name> ::= <identifier>

<link_definition> ::=
  link <link_name>
    from <external_variable_reference>
    to <external_variable_reference>;
  !link <link_name>
    <external_variable_reference>,
    <external_variable_reference>;

<link_name> ::= <identifier>
```

```
<external_variable_reference> ::=
  <object_name>.<external_variable_name>
  !<external_variable_name>

<external_variable_name> ::= <identifier>

<behavior_definition> ::=
  <equation_system_definition>!<behavior_chart_definition>

<equation_system_definition> ::=
  equations "{"
    {<equation>}
    unknown <unknown_variables>;
    "}";
  [initial "{"
    {<first_derivative_initial_value>}
  "}";]

<unknown_variables> ::=
  <variable_name>{,<variable_name>}

<first_derivative_initial_value> ::= <value>

<behavior_chart_definition> ::=
  bchart "{"
    {<state_definition>}
    {<transition_definition>}
  "}";

<state_definition> ::=
  <ordinary_state_definition>
  !<special_state_definition>

<ordinary_state_definition> ::=
  state <state_name>
    [do <local_activity_definition>]
    [<entry_actions>]
    [<exit_actions>];

<state_name> ::= <identifier>

<local_activity_definition> ::=
  <object_name>:<class_name>
  [<actual_initial_values>];

<entry_actions> ::= entry <actions_definition>;

<exit_actions> ::= exit <actions_definition>;

<actions_definition> ::=
  actions "{"
    <operator_sequence>
  "}"

<special_state_definition> ::=
  [<state_stereotype>] state <state_name>;

<state_stereotype> ::=
  initial!final!switch!switch probabilistic

<transition_definition> ::=
  <triggerful_transition_definition>
  !<triggerless_transition_definition>
```

```
<triggerful_transition_definition> ::=
  [inner] transition <transition_name>
  from <state_name> to <state_name>
  <trigger_definition>
  [<guard>]
  [<actions_definition>];

<transition_name> ::= <identifier>

<trigger_definition> ::=
  when <conditional_expression>
  ! when <signal_name>
  ! after <double_expression>

<signal_name> ::= <identifier>

<guard> ::=
  guard <conditional_expression>

<triggerless_transition_definition>::=
  transition <transition_name>
  from <state_name> to <state_name>
  [<fire_condition>]
  [<actions_definition>];

<fire_condition>::=
  "["<expression>"]"!"["else"]"

<global_constant_definition> ::=
  [<visibility>]<constant_definition>

<global_function_or_procedure_definition> ::=
  [<visibility>]<function_or_procedure_definition>

<declared_type> ::=
  [<visibility>] type <type_name> is
  <type_definition>;

<type_name> ::= <identifier>

<type_definition> ::=
  <distinct_type_definition>
  !<regular_type_definition>
  !<enumerated_type_definition>
  !<signal_type_definition>
  !<record_type_definition>
  !<connector_type_definition>

<distinct_type_definition> ::= <base_type_name>;

<base_type_name> ::= <type>

<regular_type_definition> ::=
  <vector_type_definition>!<matrix_type_definition>

<vector_type_definition> ::=
  vector ["["<static_expression>"]"];

<matrix_type_definition> ::=
  matrix ["["<static_expression>"]",
    "["<static_expression>"]"];

<enumerated_type_definition> ::= (<literal>{,<literal>};
```

```
<literal> ::= <identifier>

<signal_type_definition> ::=
  signal [(<signal_parameter_definition>
    {;<signal_parameter_definition>})];

<signal_parameter_definition> ::=
  <parameter_name>:<parameter_type>;

<parameter_name> ::= <identifier>

<parameter_type> ::= <type>

<record_type_definition> ::=
  record
    (<record_component_definition>}
  end record;

<record_component_definition> ::=
  <component_name>:<component_type>;

<component_name> ::= <identifier>

<component_type> ::= <type>

<connector_type_definition> ::=
  connector
    (<connector_component_definition>}
  end connector;

<connector_component_definition> ::=
  [<component_stereotype>]
  <component_name>:<component_type>;

<component_stereotype> ::=
  input!output!flow!contact!connector

<Model_class_definition> ::= <class_definition>

<actual_initial_values> ::=
  <variable_name>=<value>
  {,<variable_name>=<value>}
```

In this syntax:

`<identifier>` is an identifier, see **Section 6.2**.

`<type>` is any type allowed in the current context. For more information, see **Section 2.13**.

`<value>` is a value being assigned to an object (parameter, variable, constant, etc). The type of the value must be convertible to the type of the object.

`<expression>` is an expression, see **Section 2.15**.

`<text_until_end_of_line>` is an arbitrary text until end of line.

The MVL language also includes the `pragma` operator that is used to represent behavior chart and structure diagram graphics.